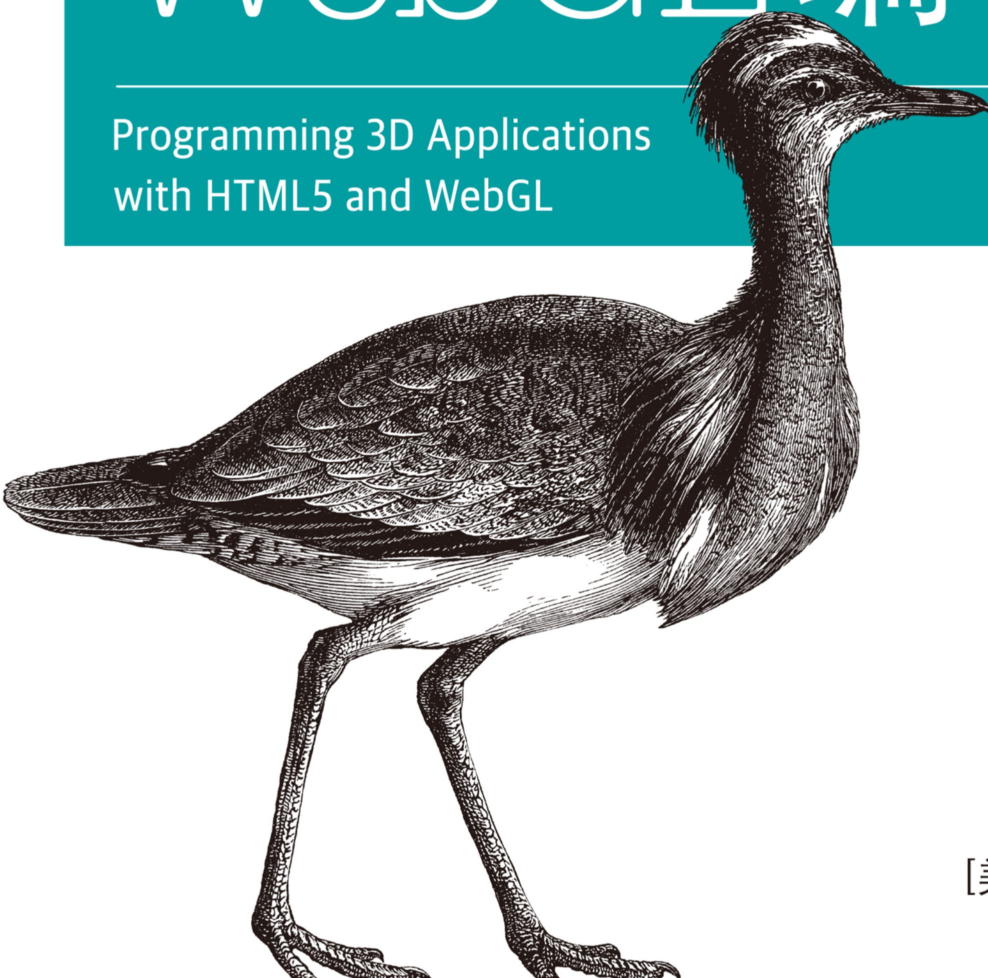
O'REILLY

TURING 图灵程序设计丛书

HTML5与WebGL编程

Programming 3D Applications with HTML5 and WebGL



[美] Tony Parisi 著
潘征 译

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

译者介绍

潘征

网名此方，目前就职于百度FEX前端研发团队，专注前端复杂应用研发，http://h5.baidu.com平台研发团队核心成员之一。

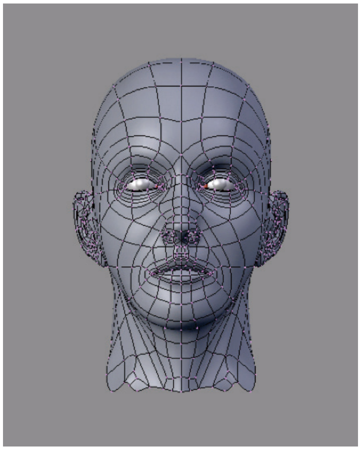


图 1-4：一个 3D 网格；遵循知识共享署名－相同方式共享 3.0 未本地化版本协议使用

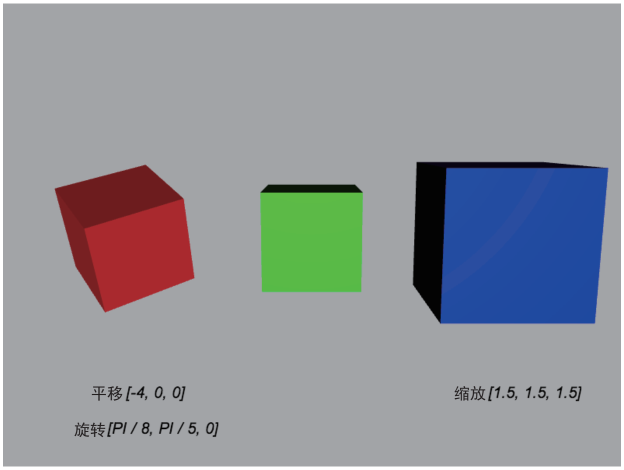


图 1-5：3D 变换——平移、旋转和缩放

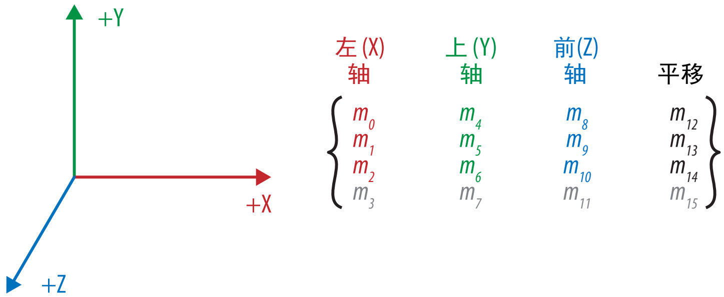


图 1-6：一个 4×4 变换矩阵（http://www.songho.ca/opengl/gl_transform.html）；经许可进行了修改

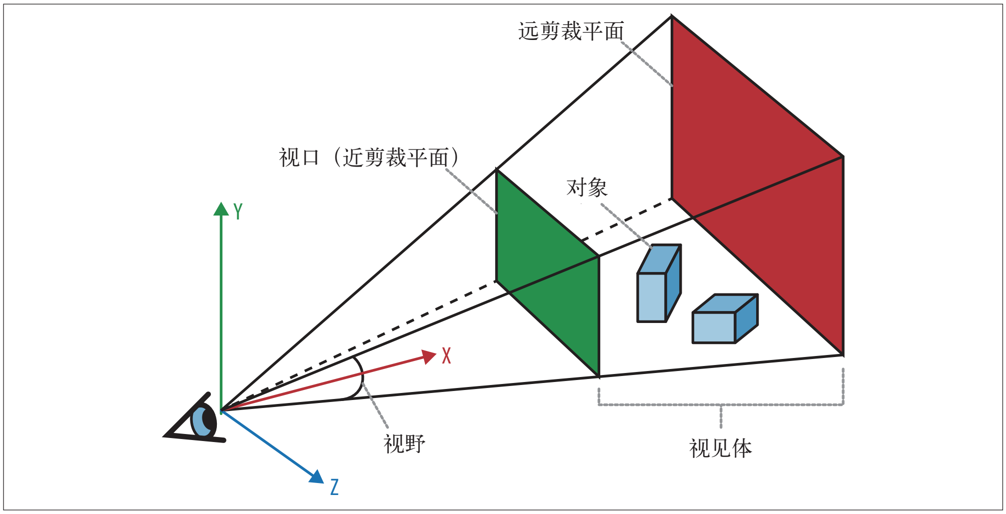


图 1-7：相机、视口和透视（http://obviam.net/index.php/3d-programming-with-android-projections-perspective/）；经许可进行了修改

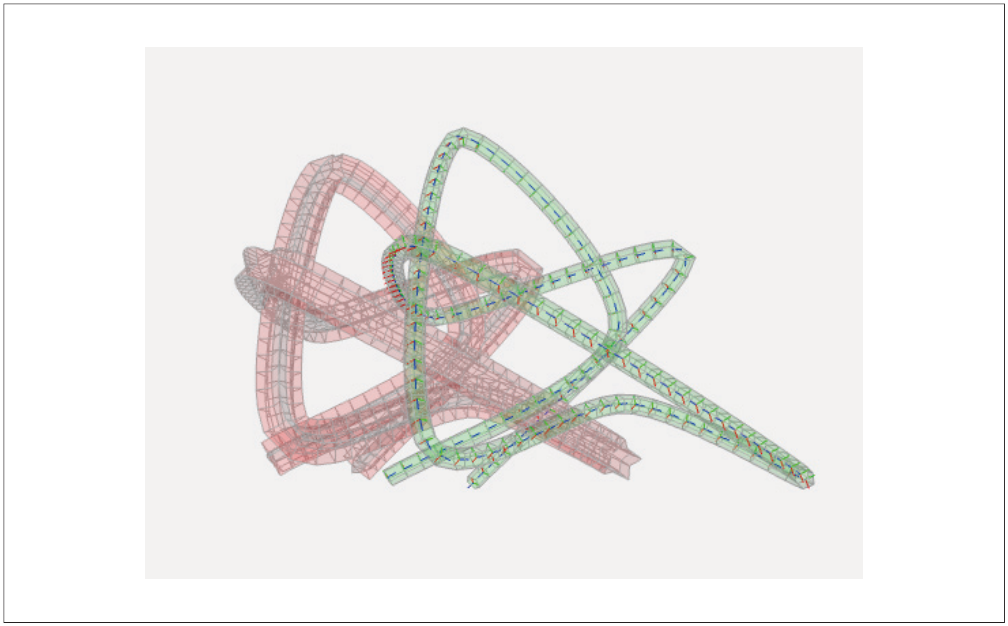


图 4-2：Three.js 中基于样条的挤出

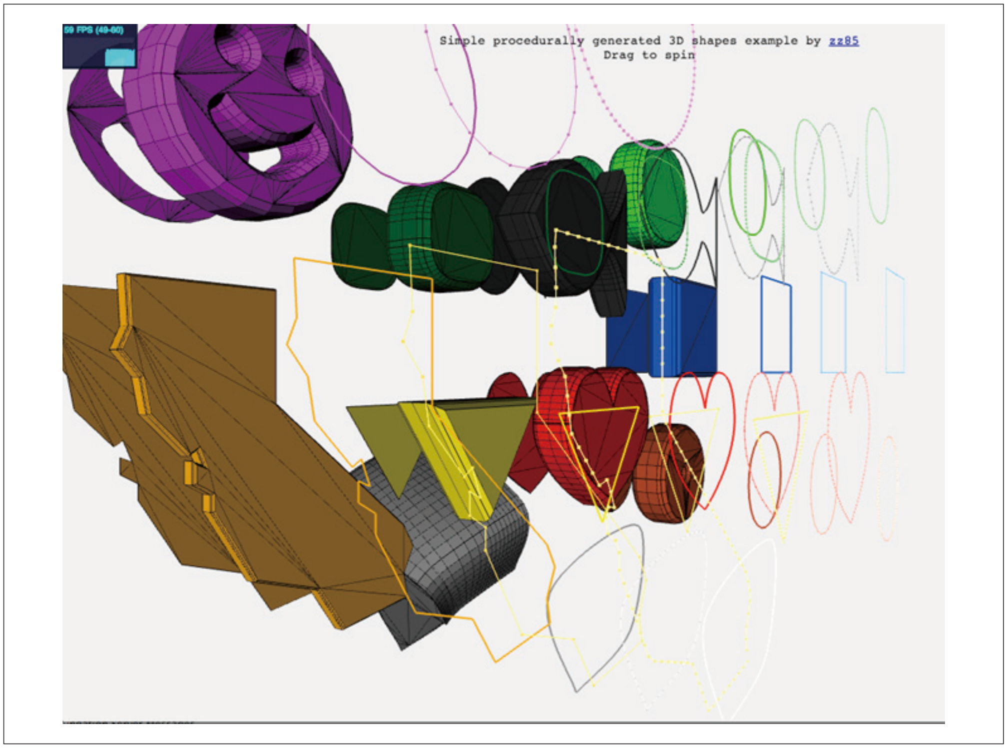


图 4-3：Three.js 实现基于路径的挤出形状

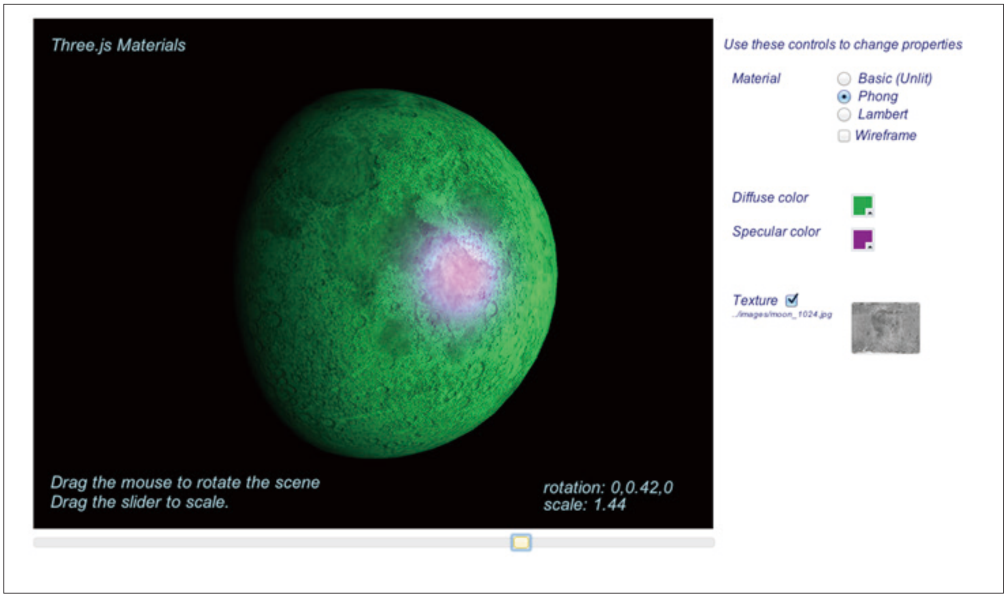


图 4-8：Three.js 标准网格材质类型——Basic (unlit)、Phong 和 Lambert

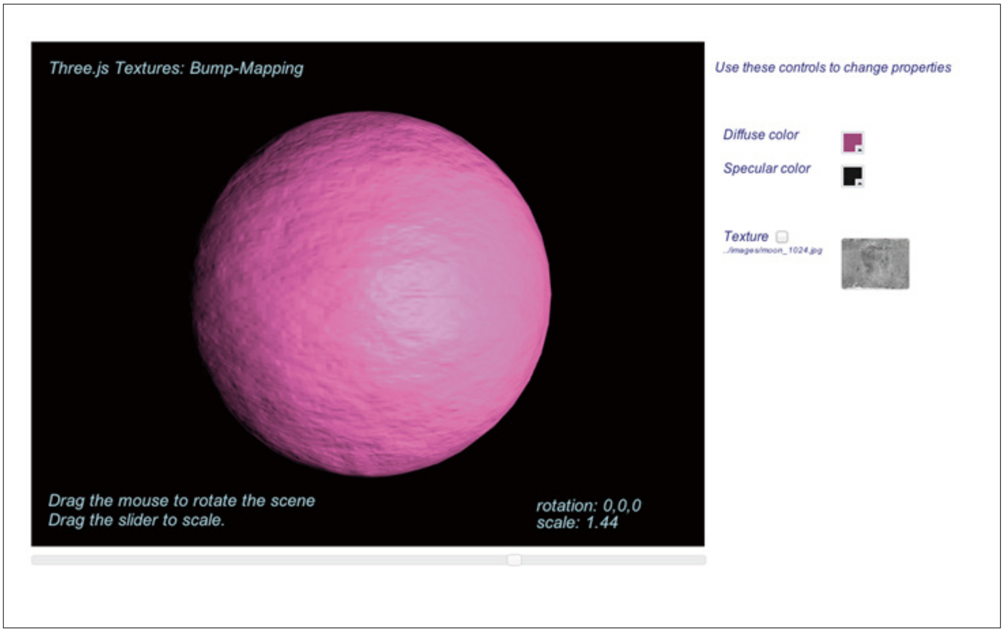


图 4-9：凹凸贴图

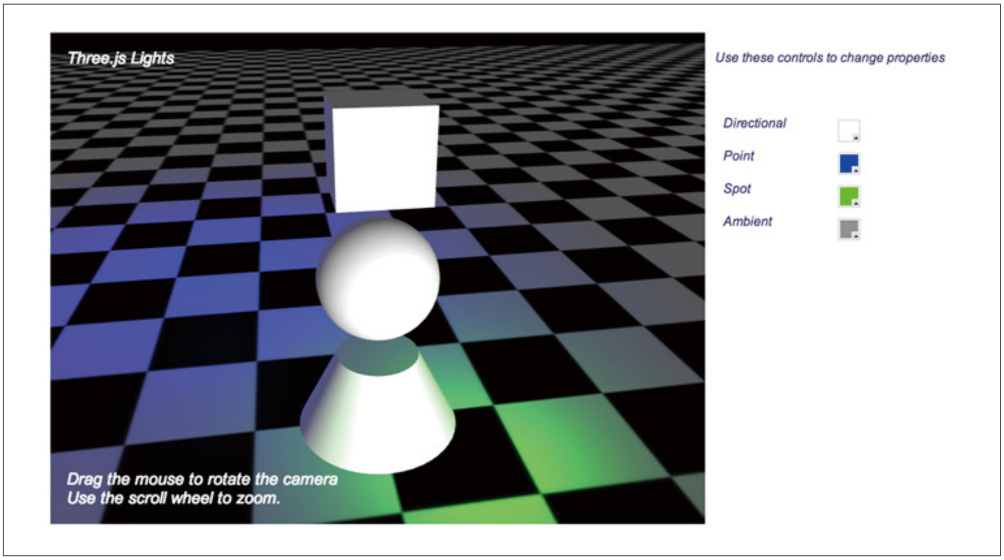


图 4-12：定向光、点光源、聚光灯和环境光

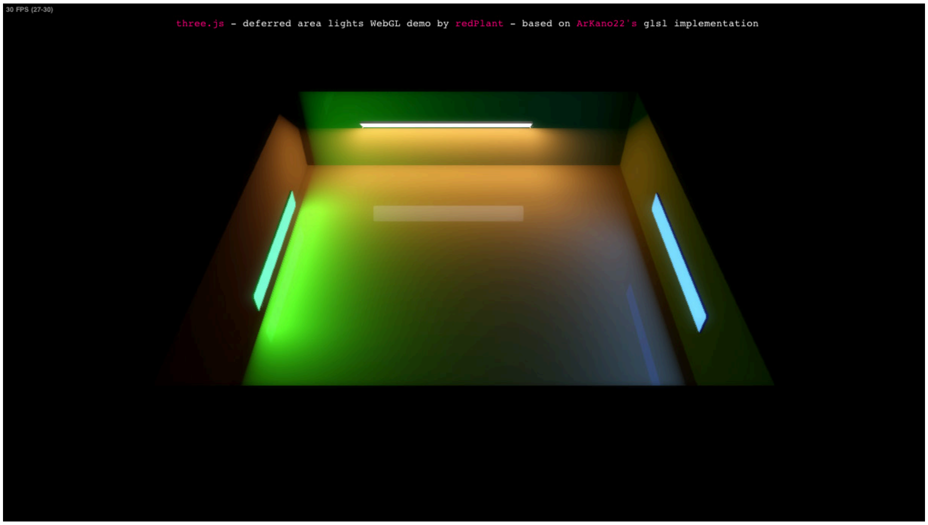


图 4-16：使用延迟渲染的逐像素光线处理

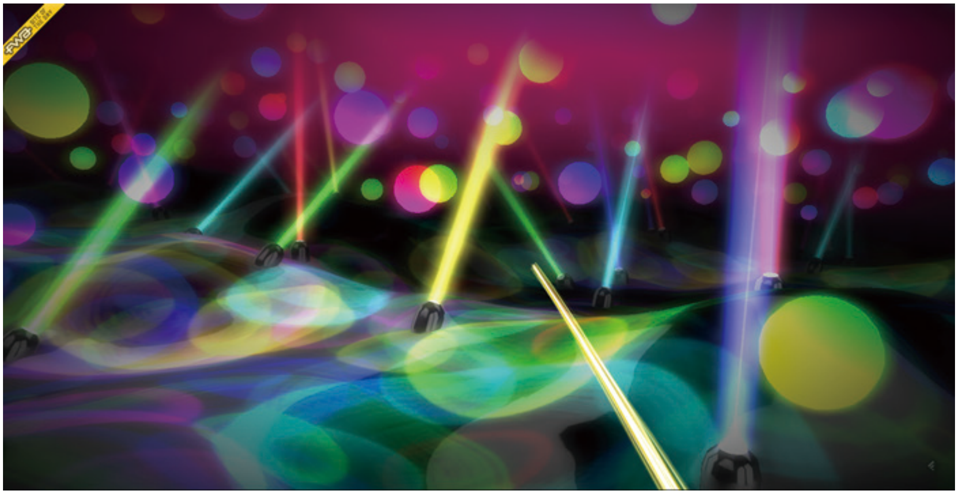


图 5-1：Ellie Goulding's Lights——一个用程序动画创建的音乐可视化项目；图片由 Hello Enjoy 提供

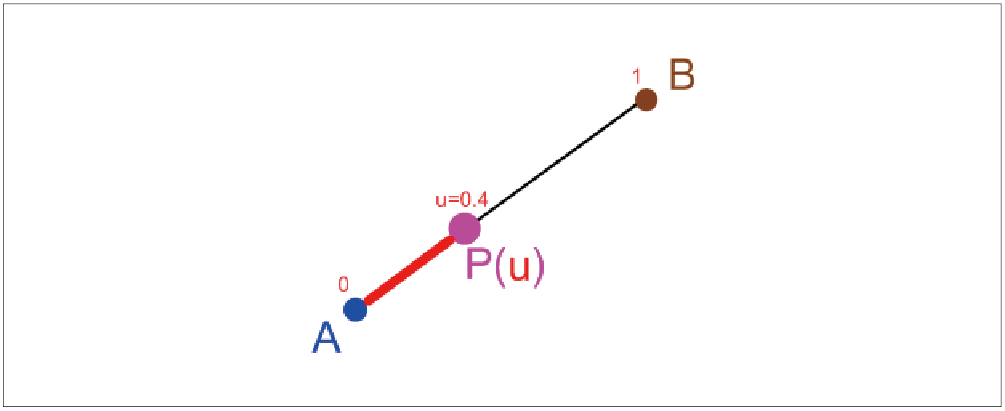


图 5-2：线性插值；转载已被许可

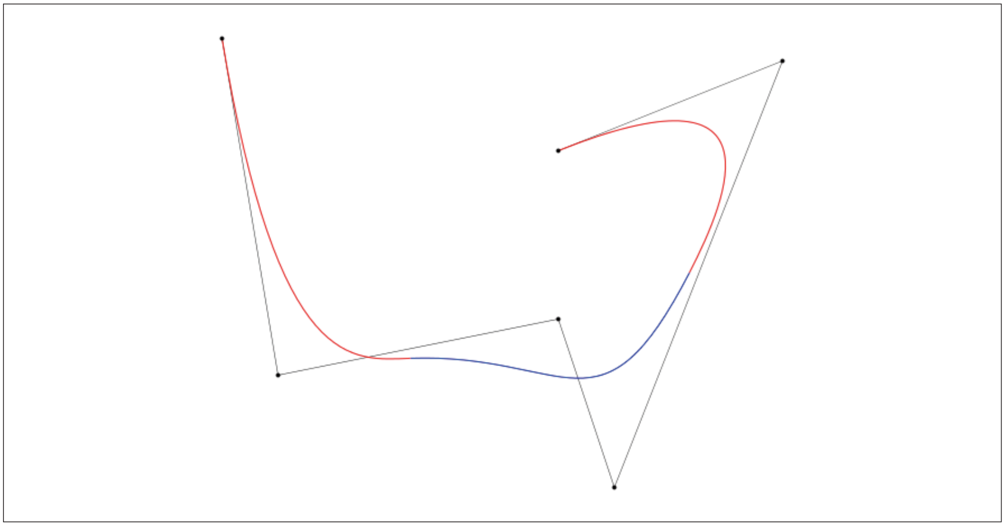


图 5-6：一个 B 样条曲线，由 Wojciech mula 提供（遵循知识共享 CC0 1.0 通用公共领域贡献协议使用）

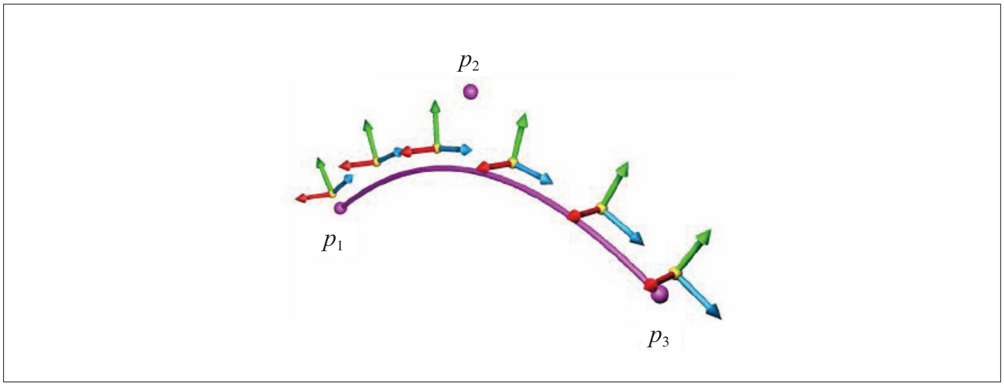


图 5-8：样条动画的坐标帧；切线、法向量和副法线分别用蓝色（向前）、绿色（向上）和红色（向右）箭头来表示；图片由 Cedric Bazillou 提供，经许可进行了修改

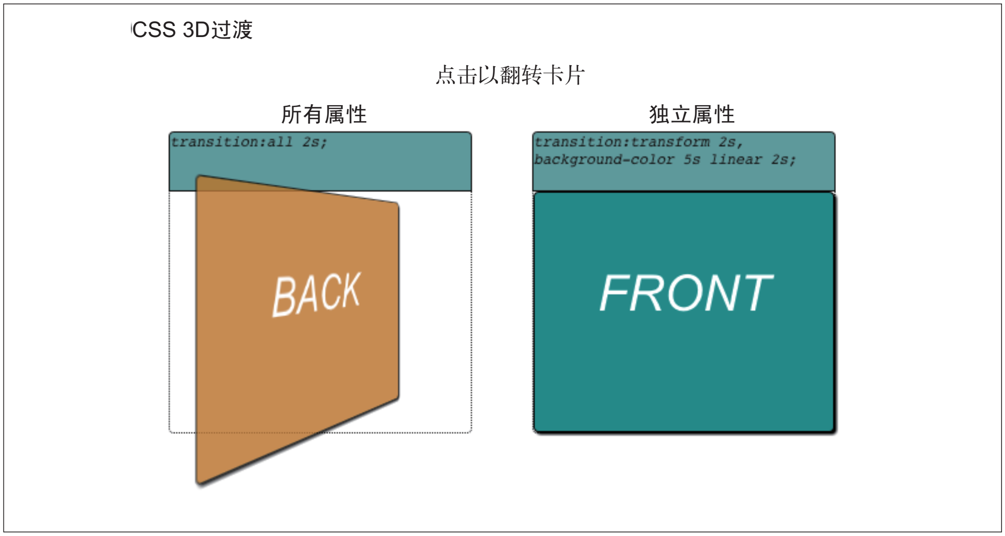


图 6-8：使用 CSS 过渡来进行属性动画

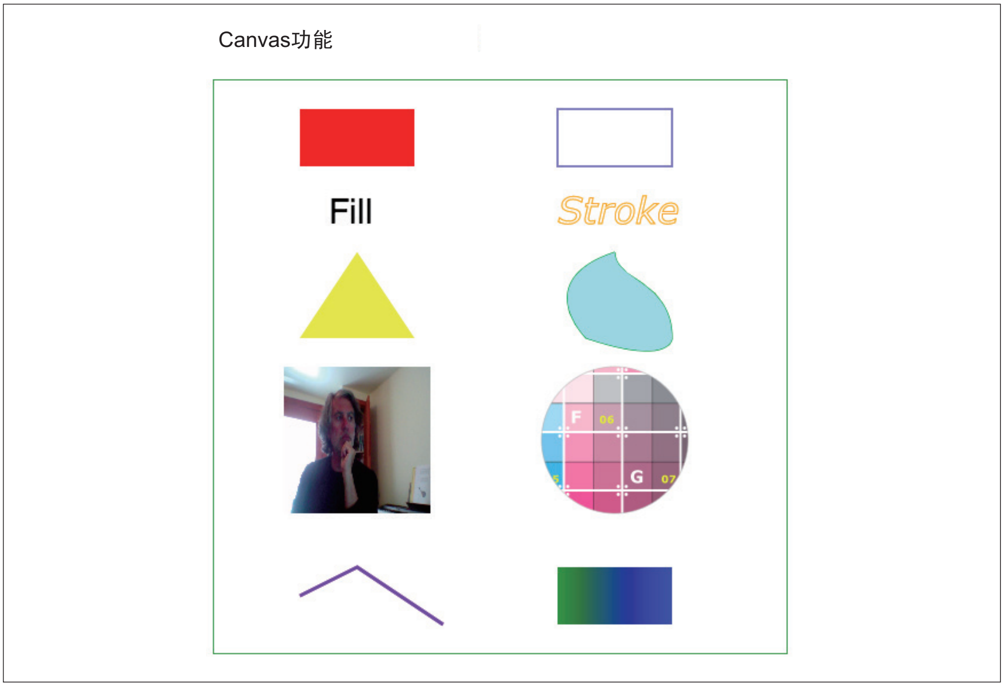


图 7-2：Canvas API 的功能

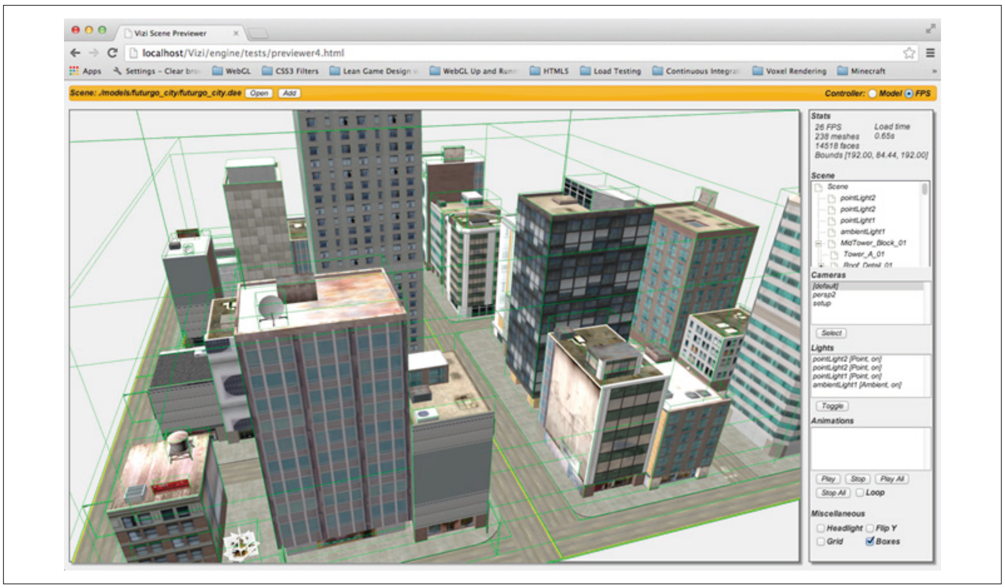


图 11-5：预览工具显示场景中所有对象的边界框

TURING 图灵程序设计丛书

HTML5与WebGL编程

Programming 3D Applications with HTML5 and WebGL

［美］Tony Parisi 著
潘征 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　　京

图书在版编目（CIP）数据

HTML5与WebGL编程 / (美) 帕里西 (Parisi,T.) 著 ; 潘征译. -- 北京 : 人民邮电出版社, 2016.6
（图灵程序设计丛书）
ISBN 978-7-115-42133-3

Ⅰ. ①H… Ⅱ. ①帕… ②潘… Ⅲ. ①超文本标记语言－程序设计②网页制作工具－程序设计 Ⅳ. ①TP312 ②TP393.092

中国版本图书馆CIP数据核字(2016)第077371号

内 容 提 要

本书全面讲解了使用 HTML5 和 WebGL 开发 3D 应用的 Web 技术，理论与实践相结合，涵盖桌面和移动两端。全书分两部分：基础知识和应用开发。在详细介绍开发相关理论、工具、框架和库的基础上，作者通过 3D 产品浏览器、游戏和交互培训系统等案例，生动讲解了 3D 应用开发的全过程。

本书适合中高级 Web 及页游开发人员阅读。

◆ 著　[美] Tony Parisi
译　潘　征
责任编辑　岳新欣
执行编辑　杨　琳　李舒扬
责任印制　彭志环

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：20.5　彩插：4页
字数：487千字　2016年 6 月第 1 版
印数：1－3 500册　2016年 6 月北京第 1 次印刷

著作权合同登记号　图字：01-2015-8286号

定价：79.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

第二部分 应用开发技术

前言

在其大约二十年的历史中，Web 3D 技术的发展过程历经曲折。1994 年，VRML 曾被视为令业界瞩目的新星，但它最终在第一次互联网泡沫中发展成了背离主流 Web 开发技术的怪胎。2000 年左右，Shockwave 3D 作为新一代备受热捧的技术，曾试图引导游戏开发界的改革。然而到了 2004 年，这项技术也同样被抛弃了。2007 年，虚拟现实系统“第二人生”（Second Life）超越科技媒体的范畴登上了《商业周刊》的封面，从而掀起了一场新的“3D 土地掠夺”运动——正如字面上的意思那样，蜂拥而来的人们租赁“第二人生”中的岛屿，试图在网络世界中创造虚幻的殖民地。而到了 2010 年，虚拟世界对人们来说已经不再新鲜，消费者们更多地依赖社交游戏和手机游戏来满足他们的消遣娱乐需求。一方面，这可以说是一连串的失败，但从另一方面来说，这也是对 Web 3D 技术的一系列锤炼。

一个好的想法可能需要耗费很长的时间来实现，但它绝不会彻底消亡。在 Web 上实现 3D 的概念也是如此。当你回顾历史——包括但不仅仅指那些早期的尝试，就能得出我们中的一些人（绝无骄傲之意）一直都知晓的理念：3D 只是一种媒体形式。无论你是用它来创建大型多人在线游戏，还是一个可交互的化学课件，或者是其他类型的应用，3D 只是让像素随着用户指示运动的另一种方式。所幸新一代的浏览器开发者们终于意识到了这一点，并逐渐将 Web 浏览器往富媒体开发平台的方向推进，包括实现了一流的硬件加速的图形渲染和一体化的图像合成架构。简而言之，3D 来了，请习惯它。

本书将介绍创建桌面和移动设备浏览器端产品级 3D 应用所需的相关信息，其中会使用现代浏览器中已经支持的一些技术：WebGL、Canvas 和 CSS3。本书涵盖的话题包括 JavaScript 性能、移动端开发以及高性能 Web 设计，并深入讲解了一些能够提高生产效率的工具和库：Three.js、Tween.js、新的应用框架以及其他创建 3D 内容的可选工具。

我的第一本书《WebGL 入门指南》（*WebGL Up and Running*，http://shop.oreilly.com/product/0636920024729.do）的读者会发现，本书的前几章和上一本书的内容有不少重叠，这是不可避免的。前几章的许多内容都是为了概述和引导，如果没有这几章的话，或许为了读懂本书你还得先读完我的上一本书。不管怎么说，尽管前几章看起来和上一本书很类似，但上一本书的读者还是能从这几章中获取到一些在上一本书中没有提及的信息。就算是介绍性章节，也比第一本书更为深入。而前三章之后的内容，会和第一本书完全不同。《WebGL 入门指南》旨在为读者提供一项也许有点令人畏惧的新技术的入门介绍。在我看

来，它或多或少缺乏技术上的严谨性，它的作用在于点燃读者对学习这门新技术的热情。如果你在读完《WebGL 入门指南》之后，产生了学习这门技术的兴趣，那么我的目的就达到了。而本书的目的在于从理论和实践两方面为读者提供更深入的内容，以实现从具备一些实践经验到真正创建产品级 3D 应用的过渡。

目标读者

本书面向尝试转向 3D 开发的有 Web 开发经验的开发者，要求读者熟悉 HTML、CSS 和 JavaScript，并至少熟悉 jQuery。3D 图形或动画方面的经验会对阅读本书很有帮助，但你并不一定要事先具备这方面的基础。本书会提供 3D 方面的入门知识，并会解释书中会用到的基本概念。

组织结构

本书分为两个部分。

第一部分，基础知识，探讨 3D 图形开发相关的底层 HTML5 API 和技术，包括 WebGL、Canvas 和 CSS3。

- 第 1 章介绍 3D 应用开发和 3D 图形的核心概念。
- 第 2 章到第 5 章深入探讨基于 WebGL 的编程，涵盖核心 API 以及 Three.js 和 Tween.js 这两个图形和动画开发的常用开源库。
- 第 6 章研究 CSS3 中用于创建 3D 页面特效和界面的新特性。
- 第 7 章讲述 2D Canvas API，以及如何在稍低端的平台上用它来模拟 3D 效果。

第二部分，应用开发技术。进入开发实战主题，包括 3D 内容的构建、使用应用框架构建程序，以及将应用部署到 HTML5 移动平台。

- 第 8 章涵盖 3D 内容的各种构建方式，包括设计师用于构建 3D 模型和动画的工具和文件格式。
- 第 9 章着重介绍如何使用开发框架来提升应用开发效率，并着重介绍了 Vizi——一个用于开发可复用 3D 组件的开源框架。
- 第 10 章和第 11 章深入介绍了几种类型的 3D 应用的开发，从通过一个可交互物体来控制动画和互动的简单应用，到带精确导航和各种交互对象的复杂 3D 环境。
- 第 12 章探讨了在新一代支持 HTML5 的移动设备和操作系统中开发 3D 应用的相关话题。

排版约定

本书使用了下列排版约定。

- *楷体*

  表示新术语。

- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。
- 加粗等宽字体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。
- 斜体等宽字体（*constant width bold*）
  表示占位符，其内容应当被用户自定义的值或上下文决定的值所替换。

> 该图标表示一般注记。

本书的示例文件

你可以从 GitHub 上下载本书的所有示例代码，网址是：

https://github.com/tparisi/Programming3DApplications

注意，你得通过一个 Web 服务器来访问本书的大部分示例，而非直接使用 file:// URL 在桌面上打开它们。这是由于 JavaScript 代码中加载了一些额外的资源文件，例如 JPEG 或 PNG 格式的图片文件；由于 WebGL 安全模型中的跨域访问安全限制，这些资源文件必须通过 HTTP 从 Web 服务器传输到浏览器端。

我在 MacBook 上运行了一个标准本地版 LAMP 环境，不过你需要用到的仅仅是 LAMP 的一部分功能——像 Apache 这样的 Web 服务器。或者如果你的机器上装了 Python，你也可以利用 Python 内置的 SimpleHTTPServer 模块来启动一个 Web 服务器，使用命令行窗口定位到 examples 目录，然后输入：

```
python -m SimpleHTTPServer
```

这样你就可以通过 http://localhost:8000/ 这个地址来访问本书的示例了。如果希望获取更多关于这方面的技术支持，请访问 *Linux Journal* 网站（http://www.linuxjournal.com/content/tech-tip-really-simple-http-server-python）。

示例文件提供了本书中创建的全部应用的完整版本，包含运行它们所需的所有文件。在某些示例中，你需要先下载一些额外的资源，例如 3D 模型资源，才能正确地运行它们。具体请查阅示例根目录中的 README 文件。

> 注意，本书中包含的许多资源受到版权限制。它们的创作者授权我将其作为本书编程示例支持，但仅可用于这一用途。如果你希望将这些代码用于其他目的，尤其是在你的应用中使用这些代码，那么你需要自行获取这些资源的许可，对于某些资源，你可能需要付费购买其许可证。

使用示例代码

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无需联系我们获得许可。比如，用本书的几个代码片段写一个程序就无需获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："*Programming 3D Applications with HTML and WebGL* by Tony Parisi (O'Reilly). Copyright 2014 Tony Parisi, 978-1-449-36296-6."

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://oreil.ly/program-3d-apps-html5-webGL

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

本书是许多人努力的结果，其中不乏许多知名人士的帮助和支持。首先我想要感谢 O’Reilly 团队。我的编辑 Mary Treseler 是一名了不起的教练，她帮助我应对了第二本书创作过程中的许多挑战。本书的写作花费了将近一年的时间——这在互联网发展中可以算是很长的一段时间了——正因如此，在写作过程中我不得不随着技术发展反复地修改书的内容，以满足读者的需求。感谢 Mary 在此过程中无与伦比的耐心和支持。技术编辑 Brian Anderson 花费大量时间为本书的结构和流程提供了大量有用的建议，助理编辑 Meghan Connolly 在将我的 Word 原稿转换为 O’Reilly 出版格式的过程中展现了精湛的技术。

我还要感谢 Ray Camden、Raffaele Cecco、Mike Korcynski 和 Daniel Smith 为本书所做的出色的技术审阅。他们详细的评论帮助我澄清了许多概念，使得示例程序更加完善。同样重要的是他们积极的反应，这保证了我对写作材料的正确使用。

书中的许多内容涉及创建 3D 图形相关的程序。在此我要向我的艺术指导 TC Chang 致以无尽的感激，他帮助我完成了第 10 章到第 12 章中展示的 Futurgo 概念车。这无疑是本书非常夺目的一个亮点。我还要感谢允许我在本书示例中使用他们的工作成果的艺术家们。你可以在 README 以及每个示例的 HTML 文件和 JavaScript 文件中找到详细的艺术家名单。在此特别感谢 TurboSquid 的技术支持负责人 Christell Gause，多亏他的费心努力，我才得到了 TurboSquid 艺术家们的授权，使得他们的创作成果得以使用在本书中。

我们很幸运地得到了 3D Web 开发者社区的强力支持。感谢 Three.js 团队所做的开创性的工作，尤其是其创始人 Ricardo Cabello (Mr.doob)。Google 的 Ken Russell 和 Brandon Jones 是世界标准的 WebGL 的实现者，但是就算再忙，他们也会耐心地回答我的问题，为我解释了 API 设计方式的缘由，并分享了未来技术的发展方向。除了 WebGL，CSS 3D 和 2D canvas 也生机勃勃。David DeSandro、Keith Clark 和 Kevin Roast 在这些领域进行了突破性的研究，并且慷慨地允许我引用他们的成果。我还要为我的好友 Don Olmstead 叫好，他几年前和我进行的一系列设计讨论最终沉淀为 Vizi，我的新 3D 开发框架。这个框架在本书中占据了重要地位。

最后要感谢我的家人。我在写作本书的同时还做着全职工作，此外还要应付其他一些事情，在此期间他们表现出了极大的耐心。Marina 和 Lucian，我欠你们一个假期，或者说三个假期。

第一部分

基础知识

第 1 章

绪论

我们生活在 3D 世界中。人们的运动、思考和感受都是 3D 的。

大多数媒体是 3D 的，尽管它们通常被展示在平面屏幕上。计算机生成的 3D 图像组成了动画电影；在线地图服务使我们能够在虚拟的 3D 环境中探索目的地；大多数电子游戏，不管是运行在专用游戏机上还是手机上，都是用 3D 渲染的；就连新闻都 3D 化了：为了在 24 小时滚动播出的新闻中吸引眼球，几年前美国有线电视新闻（CNN）的一位分析师在虚拟场景中手舞足蹈的滑稽场面，如今在有线频道的播出中已经司空见惯。

计算机 3D 图形的历史可以追溯到 20 世纪 60 年代，几乎和计算机本身的历史一样长。它被广泛应用于工程、教育、培训、建筑、金融、销售、市场、博彩、娱乐等各个领域。曾经，3D 图像只能用高端计算机系统搭配昂贵的软件来渲染，而过去十年中，这种情况已然改变。如今，所有的计算机和移动设备都搭载了 3D 图形处理硬件，普通智能手机甚至有着比十五年前的专业图形工作站更为优秀的图形处理能力。更重要的是现代 Web 浏览器也支持了 3D 渲染，相比过去昂贵的 3D 专用渲染软件，浏览器显然更普遍，更易于获取，并且还是免费的。

图 1-1 展示了一个名为“100 000 Stars”的、基于浏览器开发运行的 3D 银河系飞行模拟程序的局部。你可以用鼠标绕银道面[1]旋转整个星系，以及放大观看你所感兴趣的星体。每颗星星都根据其视星等[2]和颜色来渲染，并用标签注明其通用名。当鼠标移到标签上时，标签高亮显示。点击任意标签显示浮层，展示该星体的维基百科词条。点击浮层上的文字，浏览器会在新标签页里打开词条链接。100 000 Stars 的交互体验令人震撼，包括美丽的渲染

注 1：银道所在的主平面。银河系成员如恒星、尘埃云及气体等，绝大部分都对称地分布在这个平面的两侧。——译者注

注 2：指观测者用肉眼所看到的星体亮度。——译者注

效果、脉冲动画和宏伟的背景音乐，并巧妙地集成了 2D 用户界面。

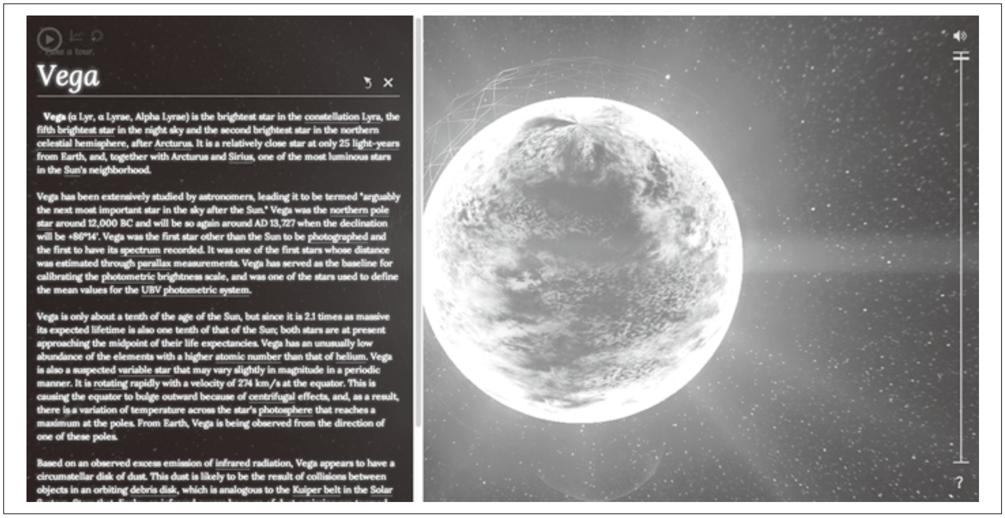


图 1-1：由 Google 创建的 100 000 Stars 项目（http://workshop.chromeexperiments.com/stars/）；图片由 Google 提供

100 000 Stars 是 Google 数据艺术小组的一个实验性项目，目的在于证明 Chrome 浏览器的丰富能力。尽管这个项目是实验性的，但它所依赖的技术却并不是：它基于被当今多数浏览器所支持的 HTML5 特性。星系和星体通过 WebGL 实时渲染，这是一项新的支持硬件 3D 加速渲染的图形标准；标签通过 CSS3 中的 3D 变换来实现与星体的位置对应；而浮层与 3D 内容可以无缝结合，则是由于所有的部件都被浏览器统一解析为页面元素来展现。

几年前，类似 100 000 Stars 这样的体验只能用本地客户端的形式来实现，用户需要下载庞大的安装包并安装到本地计算机上，而开发者们需要使用复杂的开发工具，并耗费大量的时间和金钱来实现它。现在，它可以用开源免费的工具和 Web 标准技术栈在浏览器中创建。此外，只要简单地重新加载页面，你就能立即看到最近的更新，通过 URL 在任何地方都可以加载信息，点击 3D 环境中的超链接还可以访问更多信息。

本书是关于如何利用现代浏览器的强大功能来创建在线可视化应用。其中一些应用可以看作传统 3D 产品的复刻，为覆盖更多的客户以及节省成本而使用浏览器技术来进行重构。而由于门槛的降低，各个领域的消费型应用也得以使用 3D 技术来构建——包括广告、产品营销、客户支持、教育、培训、旅游、游戏、娱乐，等等——这多么令人兴奋！ 3D 为交互体验带来了新的维度，得益于与 Web 技术的结合，全世界的人们现在都可以切身地感受这第三个维度。

> 100 000 Stars 是交互式媒体发展的力作，它的创作者之一 Michael Chang 为这个项目写了一份很好的案例分析。如果希望了解更多关于这个项目开发的事情，请访问 http://www.html5rocks.com/en/tutorials/casestudies/100000stars/。

1.1 HTML5：新型的视觉媒介

HTML 早已不像最初那样只支持静态页面、表单和提交按钮。21 世纪初，浏览器通过 Ajax 技术使得页面能够局部动态更新，从而提供了更加丰富的交互。然而用 Ajax 切换页面的具体方式始终受限于 HTML 和 CSS 的图形特性。如果开发者希望打破这些限制，就必须使用一些浏览器媒体插件，例如 Flash 和 QuickTime。

这是 21 世纪初的普遍状况，而近几年情况已经改变。这段时期，许多浏览器在发展过程中逐步支持了 HTML5。有了 HTML5，浏览器就成为一个能够运行复杂应用的平台，这些应用在功能和性能上均可媲美本地应用。HTML5 是 HTML 标准的大规模修订，包括语法的清理、新的 JavaScript 语言特性和 API、移动端支持以及突破性的多媒体支持。HTML5 平台的核心是一系列先进的图形技术，本书将重点讲述这些图形技术。

- WebGL，使得 JavaScript 支持硬件 3D 加速渲染。WebGL 基于 OpenGL，几乎所有的 PC 端浏览器都支持 WebGL，而越来越多的移动端浏览器也开始支持 WebGL。
- CSS3 3D 变换、平移以及可以支持更高级页面效果的用户自定义滤镜。经过过去几年的发展，CSS 现在已经支持硬件 3D 加速渲染和动画。
- Canvas 元素和相应的 2D 绘图 API。浏览器普遍支持这个 JavaScript 的 API，它使得开发者可以在一个 DOM 元素上绘制任意图形。尽管 Canvas 是一个 2D 绘图 API，但如果使用一些 JavaScript 库的话，它也可以用于渲染 3D 效果——在不支持 WebGL 和 CSS3 3D 的平台上，这通常被作为 3D 渲染的替代解决方案。

这些技术各有其优劣和适用场景，它们在构建 3D 可视化交互的过程中发挥着各自的作用。至于究竟要选择哪一种，你需要综合考虑多方面的因素——你想要构建什么，需要支持哪些平台，性能问题，等等。举例来说，如果你要开发一个具有高质量图像的第一人称射击游戏，若不借助 WebGL 访问图形硬件的能力，这将很难实现。又或者你在为某个视频网站开发一个有趣的调台器界面，包括当前视频缩略图、换频道的旋转效果，以及视频剪辑切换间隙的雪花噪点特效。在这个场景下，CSS3 会是创造优秀体验的良好选择。

> 总而言之，“HTML5”这个概念对大多数开发者而言指的是一系列技术和标准，包括已经被万维网联盟（W3C）采纳并被所有浏览器支持的标准，以及虽没有达到成熟标准的程度但已经被浏览器广泛支持的特性。还有如 WebGL 这类已经成熟和稳定，但不由 W3C 管理的标准。

1.1.1 浏览器平台

HTML5 为 Web 带来了丰富的图形技术；然而，若非浏览器其他特性也得到相应的发展，这些图形技术将毫无用武之地。尤其是以下提及的几点，它们为真正的 HTML5 富互联网应用的产生铺平了道路。

- JavaScript 虚拟机性能的提升

  WebGL 和 Canvas 2D 都是 JavaScript 的 API，JavaScript 的代码运行速度决定了用这些技术实现的动画和交互的运行速度。几年前，JavaScript 虚拟机的性能是 3D 技术实用

化的一大阻碍，而如今虚拟机性能的提升，使得这个问题得以解决。

- 图像合成速度的提升

  浏览器负责将页面上的元素快速合成渲染并避免不必要的重绘。随着页面上的内容更加动态，浏览器在图像合成方面有了很大的发展，包括对所有的视觉元素——不管是 2D 还是 3D——使用 3D 硬件加速渲染。

- 动画支持

  `requestAnimationFrame()` 函数是替代 `setInterval()` 和 `setTimeout()` 来驱动动画的新函数。这个新方法使得开发者能够以和浏览器刷新页面元素同步的刷新频率来更新 canvas 元素的绘图内容，这大大地提升了性能，并防止了绘图残影[3]的产生。

HTML5 浏览器同时也支持多线程编程（Web Workers）、全双工 TCP/IP 通信（WebSockets）、本地数据存储等新特性，利用它们，开发者得以构建世界级的 Web 应用。这些特性以及 WebGL、CSS3 3D 和 Canvas，结合起来象征着一个革命性的新平台，它为任何计算机和设备提供在线的可视化应用。

图 1-2 展示了 Epic Games 团队开发的 Epic Citadel 游戏的一个测试版本，它运行在 Firefox 浏览器中。Epic Citadel 使用 WebGL 来渲染图像，然而这个游戏的杰出表现，还是应当归功于游戏引擎的突破性发展。这款游戏使用了 Epic 团队实现的浏览器版 Unreal 引擎，这款引擎是本地版的 Unreal 引擎在 Web 端的移植，使用 Emscripten 编译器[4]（https://github.com/kripken/emscripten/wiki）和 asm.js（一个新的底层优化程序集）来实现。浏览器用户只需在地址栏输入一个 URL，就可以访问渲染精良的全屏游戏，游戏的刷新率可以达到每秒 60 帧（60 fps）的水准，无需安装，只需很短的加载时间。



图 1-2：运行在 Firefox 中的 Epic Citadel 测试版，一个使用 WebGL 和 asm.js 实现、刷新率达到 60 fps 的网页游戏；图片由 Epic Games 团队提供

注 3：绘图残影：http://en.wikipedia.org/wiki/Visual_artifact。——译者注

注 4：款可以将 C++ 代码编译成 JavaScript 代码的编译器。——译者注

1.1.2　浏览器支持情况

在本书写作时，3D 特性并没有被各种浏览器完全支持。不同浏览器的支持情况存在细微的差异。关于这些问题，我们将在后续章节进行详细讨论，现在先列出比较重要的几点。

- 所有的桌面浏览器都支持 WebGL。微软于 2013 年年底在 IE11 中支持 WebGL。虽然这已经落后于其他浏览器厂商，不过看起来微软会迅速赶上。
- 几乎所有的移动端浏览器都支持 WebGL：移动版 Chrome（Android）、移动版 Firefox（Android 和 Firefox OS）、Amazon Silk（Kindle Fire HDX）、Intel 的新操作系统 TIzen、BlackBerry 10。移动版 Safari 以一种受限的方式支持 WebGL（仅在 iAds 框架中）。
- 所有的浏览器和移动平台都支持 CSS 3D 变换。而 CSS 自定义滤镜仅仅被桌面版 Chrome、桌面版 Safari、移动版 Safari、BlackBerry 10 以实验性的方式支持，IE 和 Firefox 均不支持此特性。

显然，目前浏览器对 3D 特性的支持并没有达到最佳状况，然而它是随着 Web 应用这个领域的兴起而发展的。浏览器兼容性问题向来为人诟病，随着 HTML5 特性以及设备和操作系统的飞速发展，这个情况并没有得到改善。值得安慰的是，本地应用的开发、测试、部署和发布更困难，相比之下 Web 开发还算好的。这就是 21 世纪 Web 开发者的现状。

> 如果这些标准都被支持了，那么所有代码都只需写一次就行。然而，现状并非是“一次编写，随处运行”，而是“一次编写，随处调试”。

1.2　3D图形的基础知识

本节介绍 3D 图形的核心概念和术语。如果你仅仅有 2D Canvas 绘图和动画开发经验，请花一些时间去熟悉你以往没有接触过的概念，这些概念将贯穿本书。如果你有 3D 或 OpenGL 开发的相关经验，那么可以跳过本章，直接进入下一章。

1.2.1　什么是3D

既然你拿起这本书，就说明你至少对我所使用的“3D 图形”（3D graphics）这个术语有基本的认知。然而为确保你对这个概念有足够清晰的认识，我们来看看它正式的定义。以下是维基百科上“3D 计算机图形”（3D computer graphics）的条目（http://en.wikipedia.org/wiki/3D_computer_graphics）：

> 3D 计算机图形（相对 2D 计算机图形而言）是使用三个维度来表示几何数据（通常使用笛卡儿坐标系）并将其存储在计算机中，用于计算和绘制成屏幕上 2D 图像的一类图形。这类图形可被存储起来以供随时浏览，也可用于实时显示。

这个定义可展开为几个部分：(1) 数据使用三维坐标系统表示；(2) 它最终被绘制（*渲染*）为一个 2D 图像（正如你在计算机显示器上看到的那样）；(3) 它支持实时显示，即当 3D 数

据由于动画或用户操作发生改变的时候，图像可以在基本无延迟的情况下更新渲染。最后一点是创建交互式应用的关键。它非常重要，以至于催生了一个价值数十亿美元的、致力于发展支持实时 3D 渲染的图形专用硬件的产业。像 NVIDIA、ATI、Qualcomm 这些公司都是这个领域的佼佼者，对它们，你或许早有耳闻。

3D 图形并不依赖于像轨迹球、操纵杆这些特殊的输入设备，这在上文的定义中没有提及，但同样非常重要。它也不依赖于特殊的显示设备，立体眼镜和 OmniMax 电影院的昂贵门票都不是必需的。3D 图形通常被渲染在 2D 平面的显示设备上。当然这并不是说 3D 图形不能被展示为可以用立体眼镜观看的立体图像，或者是在立体电视的屏幕上播放——只是说，这不是必需的。

3D 编程需要通常 Web 开发者掌握的知识以外的新知识和技能。不过只要学会一点入门知识以及学会使用正确的工具，我们的学习进展就会非常迅速。本章接下来将着重讲解 3D 编程的基本概念，这些概念会贯穿全书。本章对这些概念的说明并不全面（有些书专门详细地讲解了这些概念），仅供入门。如果你有 3D 编程经验，完全可以跳至第 2 章 。

1.2.2　3D坐标系

如果你熟悉 2D 笛卡儿坐标系，例如 HTML 文档的窗口坐标系，那对 *x* 和 *y* 的含义一定不陌生。这些 2D 坐标定义了 `<div>` 标签在页面中的位置，以及虚拟画笔或画刷在 HTML Canvas 元素中的绘图位置。3D 绘图，正如你所料，是发生在 3D 坐标系统中的。在这个系统中，一个额外的坐标 *z* 被用于描述深度（即一个物体绘制的位置距离屏幕的远近）这个概念。本书使用的坐标系统如图 1-3 所示：*x* 轴沿水平方向延伸（从左到右），*y* 轴沿垂直方向延伸，*z* 轴正方向指向屏幕外。如果你熟悉 2D 坐标系统的概念，那么理解起 3D 坐标系统来应该也不困难。

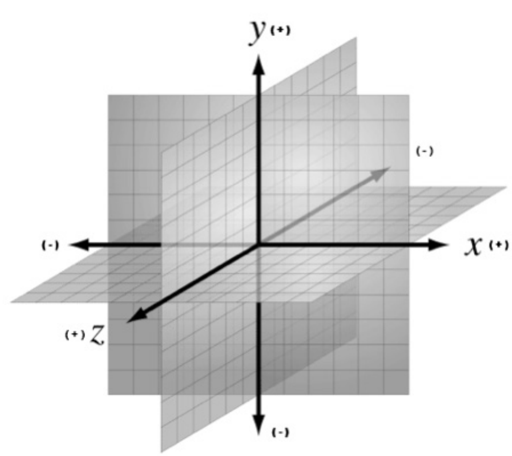
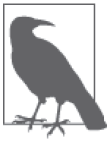


图 1-3：一个 3D 坐标系统（https://commons.wikimedia.org/wiki/File:3D_coordinate_system.svg）；遵循知识共享署名 – 相同方式共享 3.0 未本地化版本协议使用



注意，WebGL 的 *y* 轴正方向由窗口的下方向上，而 2D Canvas API 和 CSS 变换中的 *y* 轴正方向则是由上而下的。虽然这很糟，不过我们可以从中看出两种技术的传承：WebGL 基于约定 *y* 轴向上的传统图形标准，而 Canvas 和 CSS 是基于 HTML 坐标系 *y* 轴向下的约定——HTML 的这个约定则来源于早期的窗口系统坐标方案。如果你要在一个项目中同时使用这两种技术，那就得一直注意这种区别。幸好在这两种技术中 *z* 轴的方向是一致的，不然可就更麻烦了。

1.2.3 网格、多边形与顶点

绘制 3D 图形的方法有很多种。到目前为止，最常用的方法是*网格*（mesh）。一个网格通常由一个或多个多边形拼接而成，而这些多边形是由定义了 3D 空间位置（*x*, *y*, *z* 组）的*顶点*（vertice）构造出来的。在网格中，最常用的多边形是三角形（由三个顶点构造而成）和四边形（由四个顶点构造而成）。3D 网格通常简称*模型*（model）。

图 1-4 描绘了一个 3D 网格。深色线条勾勒出组成网格的四边形，定义出人脸的形状。（在最终渲染出来的图像中，你将不会看到这些线条，它们只是参考线。）网格顶点中的 *x*、*y*、*z* 属性仅仅定义了网格的形状，而网格的外观特性，如色彩和明暗等，则由其他属性来定义。下面，我们来简单地讨论这些属性。

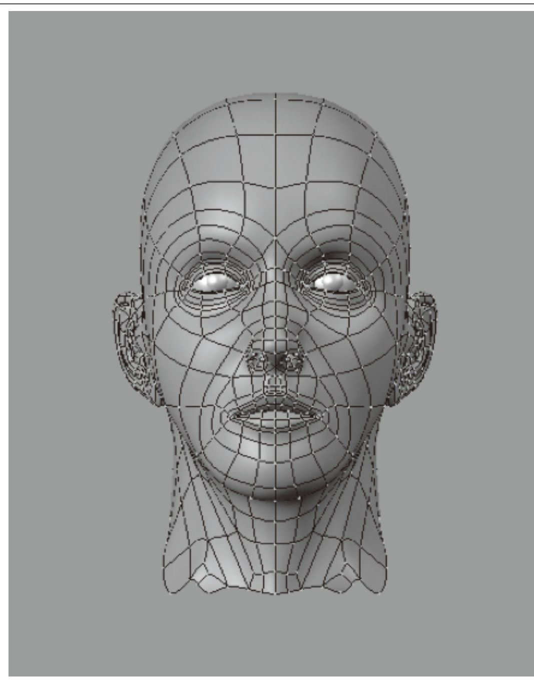


图 1-4：一个 3D 网格；遵循知识共享署名 – 相同方式共享 3.0 未本地化版本协议使用（另见彩插图 1-4）

1.2.4 材质、纹理与光源

除 *x*、*y*、*z* 这几个顶点位置信息属性之外，还有一些其他属性被用来描述网格的外观，包括简单的色彩属性和复杂的反射、明暗等属性。你还可以使用一个或多个位图来表示外观信息，这通常被称为*纹理映射*（texture map），或简称*纹理*。单个纹理可以直接定义外观样式（正如一张图像被印在 T 恤上那样），也可以与其他纹理结合起来实现复杂的效果，如表面凹凸、光的衍射等。在大多数图形系统中，网格的外观属性统称*材质*（material）。材质的展现通常依赖于一个或多个*光源*（light）的存在，这些光源定义了一个场景被照亮的模式。

图 1-4 中的人头具有紫色的表面材质，一个从模型左边发出的光源造成了模型的明暗效果。注意，暗部在脸的右侧。

1.2.5 变换与矩阵

顶点坐标的位置定义了整个 3D 网格的位置。当你希望改变一个网格在 3D 视图中的位置的时候，如果要一个个地改变每个顶点的位置，显然太麻烦了，尤其是当网格在进行持续动画的时候。为此，多数 3D 系统支持*变换*（transform）操作，即允许使用相对位置运算来移动网格，而非遍历所有顶点并确实改变它们的位置数值。变换允许你缩放、旋转以及平移（移动）一个渲染好的网格，而无需实际去改变所有顶点的数值。

图 1-5 描绘了 3D 变换操作，在这个场景中我们可以看到三个立方体。这三个立方体网格具有同样的顶点数值。当移动、旋转和缩放网格的时候，我们并没有实际去修改顶点的值，而是应用了变换。左边红色的立方体被向左平移了四个单位长度（*x* 轴上的 −4），并围绕它自身的 *x* 轴和 *y* 轴进行了旋转。（注意，旋转值的单位是*弧度*，这个单位将在第 4 章中更详细地说明。）右边蓝色的立方体被向右平移了四个单位长度，并在三个维度上都放大了 1.5 倍。中间绿色的立方体没有进行任何变换。

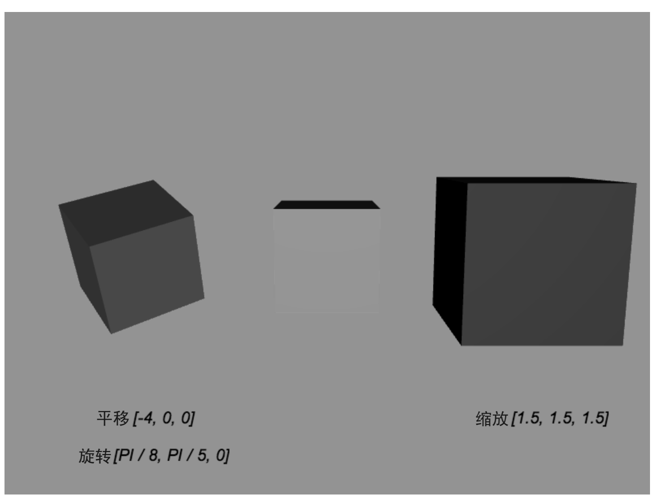


图 1-5：3D 变换——平移、旋转和缩放（另见彩插图 1-5）

3D 变换通常由一个变换矩阵（transformation matrix）来表示，这是一个包含一组用于计算转换后顶点位置的数值的运算量。绝大多数 WebGL 变换用一个 4×4 矩阵来表示——一个包含 16 个数字的、4 行 4 列的二维数组。图 1-6 展示了一个 4×4 矩阵的布局。平移被存储在元素 `m12`、`m13`、`m14` 中，分别对应 `x`、`y`、`z` 的平移值。`x`、`y`、`z` 的缩放值存储在元素 `m0`、`m5`、`m10` 中（矩阵的对角线）。旋转值存储在 `m1` 和 `m2`（*x* 轴），`m4` 和 `m6`（*y* 轴），`m8` 和 `m9`（*z* 轴）中。用这个矩阵去乘一个三维向量，便可得到变换后的值。

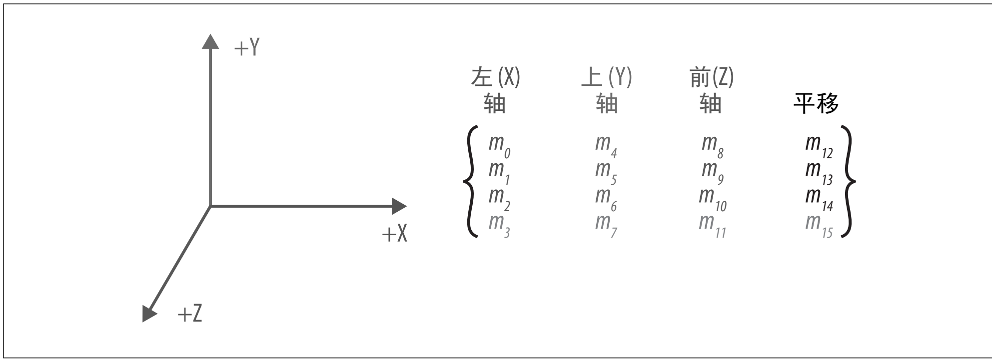


图 1-6：一个 4×4 变换矩阵（http://www.songho.ca/opengl/gl_transform.html）；经许可进行了修改（另见彩插图 1-6）

如果你和我一样是线性代数爱好者，那你会觉得用矩阵来描述变换的概念相当容易接受。就算不是也不用怕，用于构建本书示例的工具都很好地隐藏了与这些矩阵操作相关的细节：你只需以平移、旋转、缩放的直观概念来处理程序就行。

1.2.6　相机、透视、视口与投影

每个渲染好的场景都需要一个供用户查看场景的观察点。3D 系统中通常使用相机（camera）的概念来描述这个观察点。相机定义了用户相对于场景的位置和朝向，它具备现实世界中相机的属性，如视野（field of view）的尺寸，它定义了透视（perspective，即远处的物体看起来比较小）。相机的各种属性综合起来，提供了 3D 场景最终在 2D 视口（viewport）上的渲染结果，视口是由浏览器窗口或 canvas 元素决定的。

相机通常用一对矩阵来表示。第一个矩阵定义相机的位置和方向，类似变换矩阵（见前文）。第二个矩阵专门用于表示相机的 3D 坐标到视口的 2D 绘制空间坐标的转换，称为投影矩阵（projection matrix）。我就知道：讨厌的数学又来了。然而，相机矩阵的细节在多数工具中已经被很好地隐藏起来，一般来说，你只需对准、拍摄以及渲染。

图 1-7 描述了相机、视口和透视的基本概念。左下角有一个眼睛图标，它代表相机的位置。指向右侧的红色向量（在图中被标注为 *x* 轴）表示相机的指向。蓝色的立方体是 3D 场景中的物体，绿色和红色的矩形分别是近剪裁平面（near clipping plane）和远剪裁平面（far clipping plane）。这两个平面定义了一个 3D 空间子集的范围，通称视锥体或视见体（view volume 或 view frustum）。只有位于视见体中的物体才会被真正渲染到屏幕上。近剪裁平面等效于视口，在这里，我们会看到最终渲染出来的图像。

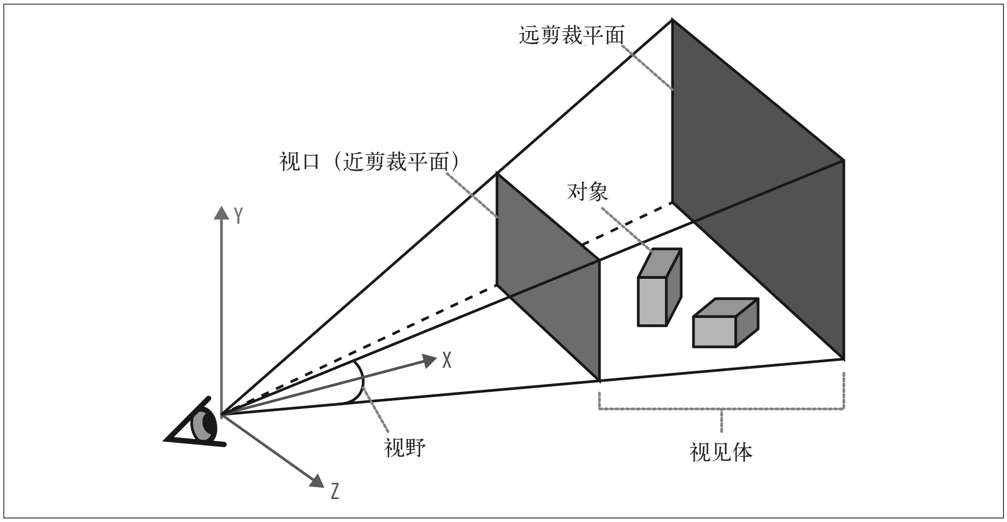


图 1-7：相机、视口和透视（http://obviam.net/index.php/3d-programming-with-android-projections-perspective/）；经许可进行了修改（另见彩插图 1-7）

相机非常强大，它从根本上定义了观察者和 3D 场景之间的关系，并为这个关系提供了现实感。相机也为动画制作者提供了强有力的武器：通过动态改变相机的位置，可以创造电影般的效果并控制其叙事风格。

1.2.7 着色器

为了渲染出一个网格的最终图像，开发者需要准确定义顶点、变换、材质、光源以及相机是如何相互作用并最终生成图像的。而承担这个工作的，是*着色器*（shader）。着色器（又称为“可编程着色器”）是一段源代码，它实现了将网格像素点投影到屏幕上的算法。图形硬件能够解析顶点、纹理以及其他底层的东西，但并不能处理材质、光源、变换以及相机。这些高级的结构由着色器程序来处理。着色器通常使用类 C 的高级语言编写，并被译成可以被图形处理单元（GPU）执行的代码。

所有的现代计算机和设备都配备了图形处理单元，一个独立于 CPU 的、专门用于 3D 图形渲染的处理器。本书中讨论的主要 3D 编程技术都以 GPU 存在为前提。

着色器为程序员的双手注入了神奇的力量：借助着色器，程序员可以精准地控制每一个像素和每一次图像渲染。着色器驱动着好莱坞电影特效中那些令人拍案叫绝的视觉效果，驱动着“CG”动画电影，驱动着现代电子游戏中的实时渲染。随着 Web 浏览器对着色器的支持，我们的 WebGL 应用效果甚至可以媲美最好的电子游戏。而利用着色器，我们也得以更好地控制页面中 CSS 元素的展现和动画。

图 1-8 展示了一个使用可编程着色器渲染的 WebGL 水波模拟案例。碧波荡漾、光影跃动，

逼真得惊人，而且你还可以实时地与场景进行交互，尽管它是虚拟的。提醒一下：这可是在一个 Web 浏览器中运行的！

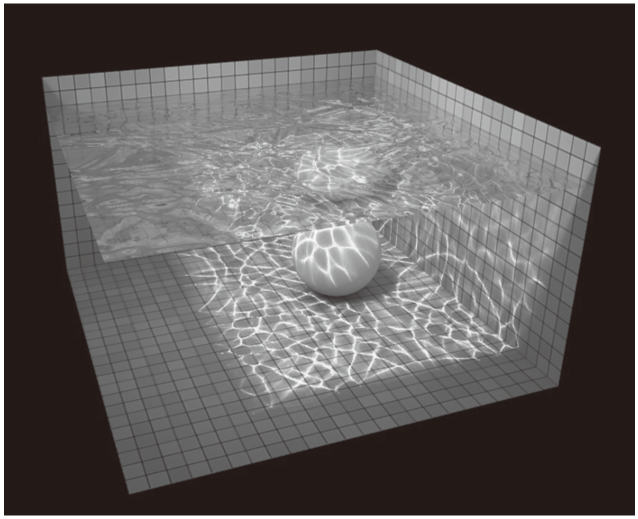


图 1-8　使用可编程着色器渲染的 WebGL 水波模拟，创作者 Evan Wallace（http://madebyevan.com/webgl-water/）；转载已被许可

除了 WebGL，借助 CSS *自定义滤镜*（CSS Custom Filter）这项实验性的技术，基于着色器实现的特效同样可以应用于 DOM 元素。关于这一点，我们在第 6 章中将会详细讨论。

以下列举了一些与着色器相关的技术点，我们将在本书中讲述。

- WebGL 和 CSS 自定义滤镜都使用 OpenGL ES 着色语言（GLSL ES）定义的着色器。在 WebGL 中编写着色器和为 CSS 编写着色器不太一样，但基本的编程语言是通用的。
- 在 WebGL 中，物体的渲染绘制依赖于着色器。如果没有提供着色器，或者编译或加载着色器过程出错，则所有物体都无法成功渲染。
- 在 CSS3 滤镜中，着色器是可配置的。当着色器被用来定义 CSS3 滤镜的时候，它被称为*自定义滤镜*。
- 2D Canvas API 不支持可编程着色器。如果你打算用 2D Canvas 作为 WebGL 的降级方案，得在代码里面自己实现这套机制。关于这一点，在第 7 章中将有详细说明。

着色器从某种意义上来说代表了一条学习曲线，你需要学习新的特性以及一门新的编程语言，这需要耗费大量的精力。如果你觉得这太可怕了，别担心，有许多流行的开源库和工具可供选择，如此一来，你就无需去关注那些艰涩的着色器具体实现细节。甚至在你的 3D 编程职业生涯中，你连一行 GLSL 代码都无需编写——不过我建议你还是试试看，这样一来，你就可以对别人说你做过这件事啦。

关于 3D 图形基础内容的介绍至此结束。书中提及的每种技术的具体实现会略有不同，但核心的概念都是一致的。在接下来的几章中，我们将深入研究如何使用 WebGL、CSS3 和 Canvas 2D 来创建 3D 内容并实现 3D 动画。

第2章

WebGL：实时3D渲染

WebGL 是 Web 3D 图形的标准 API，它使得运行在浏览器中的 JavaScript 程序也可以充分利用 3D 渲染硬件的强大能力。在 WebGL 出现之前，为提供硬件加速的 3D 体验，开发者只能借助浏览器插件或编写需要用户下载安装的本地软件。

尽管 WebGL 不属于 HTML5 官方标准，但绝大多数支持 HTML5 的浏览器都支持 WebGL——正如支持 Web Workers、WebSockets 等并未被 W3C 官方作为标准采纳的技术一样。要想将浏览器打造成一流的应用平台，3D 是不可或缺的部分，这是 Google、Apple、Mozilla、Microsoft、Amazon、Opera、Intel、BlackBerry 等各大公司开发者们的共识。

主流的桌面浏览器和绝大多数手机浏览器都支持 WebGL[1]。WebGL 已经可以运行在类似于你的家用机器和办公机器的数百万台设备上。包括游戏、数据可视化、计算机辅助设备、3D 打印和零售行业在内的许多使用了 WebGL 技术的网站也在蓬勃发展。

WebGL 是一套底层绘图 API：它通过解析数据和着色器阵列[2]来进行绘制。它不像 2D Canvas API 那样具有高度封装的结构，这可能会令习惯 2D 图形接口的人感到困惑。不过很多开源的 JavaScript 工具包都提供了更加高级的封装方法，这些工具包让开发者可以用与操作传统图形库更为接近的方式来操作 WebGL 的 API。虽然有了这些工具包，3D 开发也还是有一定的难度，但至少利用它们，对 3D 开发没什么经验的人可以比较方便地入门，而有经验的 3D 开发者也可以节省大量时间。

注 1：在本书写作时，iOS 上的 Mobile Safari 还不支持 WebGL，这是个很严重的问题。所幸利用某些适配工具包，我们可以将基于 HTML5 和 WebGL 的程序打包成本地应用在 iOS 平台上运行、研究，关于这个问题，在第 12 章中会有详细的说明。

注 2：阵列，指排成行和列的数学元素。矩阵就是一种典型的阵列形式，此处姑且可以简单理解为一个数据 / 着色器二维数组。——译者注

为了让读者对 WebGL 有个基本印象，本章将简单介绍 WebGL 的底层基础。虽然我们在本书中使用的工具包可以让你不必去关注这些底层细节，但了解这些工具包是基于什么构建的也非常重要。所以，让我们从 WebGL 的核心概念和 API 开始学习。

> 正如不支持许多 HTML5 新特性一样，你的电脑可能也不支持 WebGL。主流桌面浏览器中，一部分浏览器只有比较新的版本才支持 WebGL（例如 IE 只有 IE11 及其之后的版本才支持 WebGL）。还有一些老机器的图形处理器不支持 3D 硬件加速，在这些老机器上，浏览器会直接关闭 WebGL。如果你想了解你的目标机器、设备或浏览器是否支持 WebGL，请访问 http://caniuse.com/，键入“WebGL”关键字进行搜索，或直接访问这个链接：http://caniuse.com/#search=WebGL。

2.1 WebGL基础

WebGL 的雏形在 2006 年由 Mozilla 的工程师 Vladimir Vukićević 提出。Vladimir Vukićević 试图创建一套用于 Canvas 元素的 3D 绘图 API，作为已有的 2D Canvas API 的扩展。他基于 OpenGL ES（在移动端图形领域已经相当普及的 API 标准）设计了这套当时还被称为 Canvas 3D 的 API。到 2007 年，Mozilla 和 Opera 分别实现了各自浏览器上的 Canvas 3D 版本。

2009 年，来自 Opera、Apple 以及 Google 的其他参与者与 Vukićević 共同创建了 WebGL 工作组，这个工作组隶属于 Khronos Group 团队，Khronos Group 团队还维护着 OpenGL、COLLADA 以及其他一些你或许已经耳熟能详的标准。如今 Khronos 仍然在继续维护 WebGL 标准。Vukićević 担任 WebGL 小组负责人直到 2010 年，之后 Google 的 Kenneth Russell 接替了他。

以下是来自 Khronos 网站的 WebGL 官方描述：

> WebGL 是一套免费、跨平台的 API，它在 HTML 中以 3D 绘图上下文的形式实现了 OpenGL ES 2.0 的功能，并以底层的文档对象模型（Document Object Model，DOM）接口的形式将开发接口暴露出来。它使用 OpenGL 着色器语言 GLSL ES，并且可以与页面上的其他内容（可以分层的形式叠加在 3D 绘图区域的上方或下方）无缝融合。它非常适应于使用 JavaScript 编程语言构建的 3D Web 动态应用，并将被现代浏览器完美支持。

这个定义包括几个要点，下面我们来分别说明。

- **WebGL 是一套 API**。它通过一套专门的 JavaScript 编程接口来调用，不像 HTML 那样带有附带的标记。3D 渲染与 2D 绘制同样使用 Canvas 元素来作为绘图上下文，开发者通过对 JavaScript API 的调用，在 Canvas 中实现 3D 内容的绘制。事实上，对 WebGL 接口的访问是通过现有 Canvas 元素中的 3D 专用绘图上下文来实现的。
- **WebGL 基于 OpenGL ES 2.0**。OpenGL ES 是长久以来的 3D 渲染标准 OpenGL 的一个适配方案。ES 代表“嵌入式系统”（embedded system），这表示它专用于小型计算设备，尤其是手机和平板电脑。OpenGL ES 是为 iPhone、iPad 和 Android 手机提供 3D 图形渲

染能力的标准接口。WebGL 的设计者们认为，基于 OpenGL ES 更小的硬件占用量，可以更方便地提供一套统一的、跨平台跨浏览器的、用于 Web 的 3D API。

- **WebGL 可以与其他 Web 页面元素相结合**。WebGL 可以以分层的形式置于其他页面内容的上方或下方。3D canvas 可以占据页面的一部分或整个页面，它也可以被包含在被设置了 z-index 属性的 `<div>` 元素中。这意味着你可以使用 WebGL 来构建 3D 图形，而使用你所熟悉的 HTML 特性来构建其他页面元素，并将它们（通过浏览器）无缝组合在同一个页面上展示给用户。
- **WebGL 为创建动态 Web 应用而生**。WebGL 在设计过程中就考虑到了网络传输的需要。它始于 OpenGL ES，但它加入了许多与浏览器适配的特性。它使用 JavaScript 编写，并且对 Web 传播相当友好。
- **WebGL 是跨平台的**。WebGL 可以运行在任意操作系统上，无论是手机、平板电脑还是台式电脑。
- **WebGL 是免费的**。正如所有的开放 Web 标准那样，WebGL 可以免费使用。没有人会因为你使用了 WebGL 而要求你支付版权费用。

Chrome、Firefox、Safari 以及 Opera 的创造者们为发展和支持 WebGL 投入了众多的资源。来自这些浏览器开发组的工程师们同样也是 WebGL 标准组织的重要成员。WebGL 标准的发展进程对 Khronos 小组的全体成员开放，同时也有对公众开放的邮件组。本书的附录中提供了邮件组和其他标准相关资源的详细信息。

2.2 WebGL API

WebGL 基于成熟的图形 API——OpenGL。WebGL 始于 20 世纪 80 年代末期，在经历了来自微软的 DirectX 的竞争威胁后，成为了 3D 图形编程无可争议的行业标准。

但所有的 OpenGL 版本都不尽相同。不同平台——包括台式电脑、电视机机顶盒、智能手机以及平板电脑——具有不同的特性，因此针对不同平台的 OpenGL 版本也由此发展起来。OpenGL ES 是针对机顶盒和智能手机这类小型设备的 OpenGL 版本，因此，它成为了 WebGL 的理想核心。它小而精，这不仅意味着它可以直接被浏览器实现，更意味着不同浏览器的开发者都可以很方便地在浏览器中实现对 WebGL 的支持，从而使得在一款浏览器中编写测试的 WebGL 应用可以直接运行在另一款浏览器中。

WebGL 非常精巧，这使得 WebGL 应用的开发者们需要做更多的工作。3D 场景本身不具备 DOM 结构[3]，也没有原生支持的用于加载几何图形和动画的 3D 文件格式；除了几个底层的系统事件以外，3D canvas 并不具备内建的事件机制（例如，当你点击场景中的某个物体的时候，这个物体并不会触发鼠标点击事件）。对一般的 Web 开发者来说，WebGL 意味着陡峭的学习曲线和许多陌生的概念。

有许多开源的代码库可以让 WebGL 开发变得不那么困难，正如 jQuery 或 Prototype.js 对于传统 Web 开发的意义那样——这个比喻或许比较粗糙。在接下来的几章我们会详细介绍这

注 3：可以理解为 3D 场景中的单个物体不具备单独的文档结构，即整个 canvas 中的绘图内容被视为一个整体。——译者注

些库。不过现在我们先来简单了解一下底层的 WebGL，尽管在你的项目中你可能不会有机会去编写这些底层的 WebGL 代码，但了解一下引擎背后的实现机制，总归是有好处的。

2.3 WebGL应用剖析

说到底，WebGL 就是个绘图库，类似 2D canvas 那样，是被所有支持 HTML5 的浏览器所支持的另一种 canvas。事实上，WebGL 也使用了 Canvas 元素来作为在浏览器页面上绘制 3D 图形的容器。

为了在页面中渲染 WebGL，一个应用至少应当执行以下步骤：

(1) 创建一个 Canvas 元素；
(2) 获取 Canvas 元素中的绘图上下文；
(3) 初始化视口；
(4) 创建一个或多个包含待渲染数据（通常是顶点数据）的缓冲；
(5) 创建一个或多个定义顶点缓冲到屏幕空间转换规则的矩阵；
(6) 创建一个或多个实现绘制算法的着色器；
(7) 使用各项参数初始化着色器；
(8) 绘制。

下面我们用一些示例来说明这个流程。

2.4 一个简单的WebGL示例

为了说明 WebGL API 的基本工作机制，我们来编写一个非常简单的程序，在 canvas 上绘制一个白色正方形。文件 Chapter 2/example2-1.html 中有这个示例的完整代码，绘制结果如图 2-1 所示。

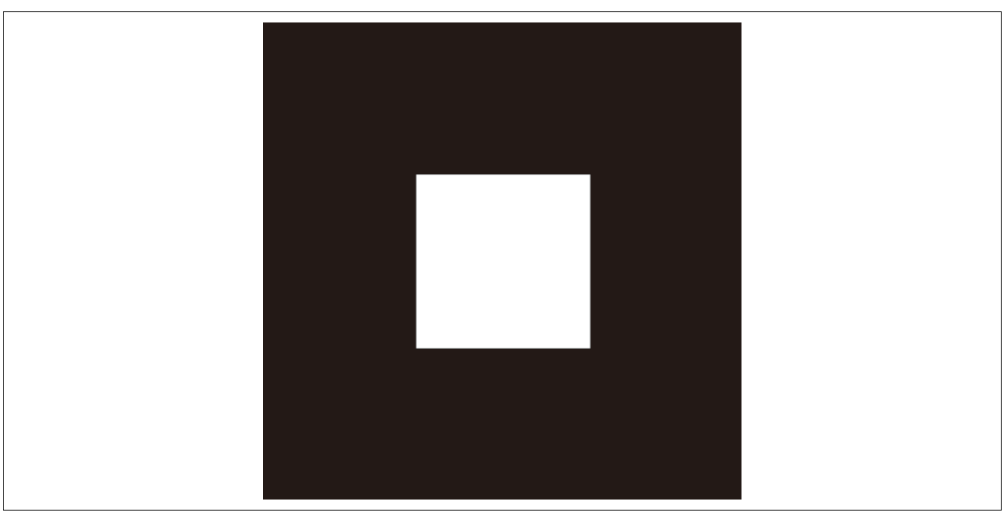


图 2-1：使用 WebGL 绘制的一个正方形

本节大部分示例的灵感来源于 Learning WebGL（http://learningwebgl.com/）上的课程。Learning WebGL 由 Giles Thomas 创立，是一个出色的资源站点，它提供了一系列学习 WebGL 的教程。该网站还设有关于 WebGL 新应用的周刊，可供随时了解 WebGL 最新的发展动向。

2.4.1 Canvas元素和WebGL绘图上下文

所有的 WebGL 渲染都发生在一个上下文（context）中，这是一个提供了完整 WebGL API 的 DOM 对象。这个结构与 HTML5 Canvas 元素提供 2D 绘图上下文的模式相同。想要在页面中插入 WebGL 的内容，首先需要在页面上的某个位置创建一个 <canvas> 标签，获取与其相关的 DOM 对象（可以使用 document.getElementById()），并获取这个 DOM 对象的 WebGL 绘图上下文。

例 2-1 展示了如何从 canvas DOM 元素中获取 WebGL 绘图上下文。getContext() 方法支持两种表示上下文 id 的字符串参数，"2d" 参数用于获取 2D Canvas 绘图上下文（在第 7 章讲述），"webgl" 或 "experimental-webgl"（较老版本的浏览器）用于获取 WebGL 绘图上下文。新的浏览器同时兼容 "experimental-webgl" 和 "webgl" 参数。在示例代码中我们使用 "experimental-webgl"，以确保它能够兼容不同版本的支持 WebGL 的浏览器。

例 2-1：从 canvas 中获取 WebGL 绘图上下文

```
function initWebGL(canvas) {

   var gl = null;
   var msg = "Your browser does not support WebGL, " +
       "or it is not enabled by default.";
   try
   {
       gl = canvas.getContext("experimental-webgl");
   }
   catch (e)
   {
       msg = "Error creating WebGL Context!: " + e.toString();
   }

   if (!gl)
   {
       alert(msg);
       throw new Error(msg);
   }

   return gl;
}
```

注意示例中的 try/catch 代码块。它非常重要，因为某些浏览器，或者某些浏览器的旧版本并不支持 WebGL。即使浏览器支持 WebGL，浏览器运行的硬件设备也有可能由于太旧而无法提供 WebGL 绘图上下文。故而上述检测代码能帮助你在适当的时机加载降级方案（例如使用基于 2D canvas 的渲染方案），或者至少优雅地退出。

2.4.2 视口

当你从 canvas 中获取到一个 WebGL 绘图上下文时，需要定义一个绘制区域的矩形边界。在 WebGL 中，这个矩形边界被称为*视口*（viewport）。在 WebGL 中，设置视口非常简单，只需调用绘图上下文的 `viewport()` 方法，正如例 2-2 中那样。

例 2-2：设置 WebGL 视口

```
function initViewport(gl, canvas)
{
    gl.viewport(0, 0, canvas.width, canvas.height);
}
```

注意这里的 `gl` 对象是由上面定义的 `initWebGL()` 函数生成的。在这里，我们将整个 canvas 区域都定义为 WebGL 的视口。

2.4.3 缓冲、缓冲数组和类型化数组

现在我们已经得到了等待绘制的 WebGL 绘图上下文。到目前为止，我们所做的事情和在 2D Canvas 上绘制图形所需的准备工作并没有多大区别。

WebGL 基于*图元*（primitive）进行图像绘制。所谓图元，是指不同类型的基本几何图形。WebGL 的图元包括三角形、点和线。三角形是最常用的图元类型，通常使用两种形式存储：三角形集（以数组形式存储的三角形）和*三角形条带*（triangle strip）[4]。图元以数组的形式存储数据，这个数组被称为*缓冲*（buffer），待绘制的顶点数据在缓冲中被定义。

例 2-3 展示了如何为一个单位（1×1）正方形创建顶点缓冲。其结果在一个包含顶点缓冲数据的 JavaScript 对象中返回，包括顶点结构的长度（在这个示例中，表示顶点 x、y 和 z 值的每组数据用三个浮点数来存储），待绘制的顶点数量，绘制这个正方形时所使用的图元类型——在这个示例中，我们使用了三角形条带。一个三角形条带定义了一组连续的三角形，前三个顶点表示第一个三角形，后续的每个三角形都与前一个三角形共用其两个顶点。

例 2-3：创建顶点缓冲数据

```
// 构建用于绘制的正方形顶点数据
function createSquare(gl) {
    var vertexBuffer;
    vertexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
    var verts = [
         .5,  .5,  0.0,
        -.5,  .5,  0.0,
         .5, -.5,  0.0,
        -.5, -.5,  0.0
    ];
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(verts), gl.STATIC_DRAW);
    var square = {buffer:vertexBuffer, vertSize:3, nVerts:4,
```

注 4：三角形条带是一种通过重复利用顶点数据来压缩数据体积的存储方式，关于这种存储方式，请参照维基百科词条 http://en.wikipedia.org/wiki/Triangle_strip。——译者注

```
        primtype:gl.TRIANGLE_STRIP};
    return square;
}
```

注意类型 Float32Array。这是一种浏览器专门为 WebGL 引入的新数据类型。Float32Array 是一类缓冲数组（ArrayBuffer），也称为类型化数组（typed array）。它是一种以二进制方式存储的 JavaScript 类型。你可以用与访问普通数组相同的方式来访问类型化数组，但访问类型化数组的速度更快，耗费的内存也更小。由于使用二进制数据存储，它们是解决性能瓶颈的理想存储方案。类型化数组可以被普遍使用（不仅仅在 WebGL 中），但它是因 WebGL 才被引入浏览器的。我们可以在 Khronos 组织的网站（https://www.khronos.org/registry/typedarray/specs/latest/）上找到类型化数组的最新规范。

2.4.4 矩阵

在绘制正方形之前，我们首先要创建一对矩阵。一个矩阵用于定义正方形在 3D 坐标系统中的位置（相对于相机），这个矩阵被称为*模型-视图矩阵*（ModelView matrix），因为它同时包含模型矩阵（模型位置）和视图矩阵（相机位置）的信息。在我们的示例中，我们沿着 z 轴负方向对正方形进行了平移（即将它移动到距离相机 −3.333 个单位长度的地方）[5]。第二个矩阵被称为*投影矩阵*（projection matrix），着色器使用它来执行从 3D 空间坐标到 2D 视口绘制空间坐标的转换。在这个示例中，投影矩阵定义了一个 45 度角视野的透视相机（如果你忘了什么是透视投影，请回顾第 1 章）。

在 WebGL 中，矩阵以类型数组形式存储的一组数字来表示。例如一个 4×4 矩阵使用一个包含 16 个元素的 Float32Array 对象来表示。为了更方便地初始化矩阵和进行矩阵运算，我们将使用一个叫 glMatrix 的开源库（https://github.com/toji/gl-matrix），这个库由现任 Google 工程师的 Brandon Jones 编写。矩阵初始化的代码如例 2-4 所示。glMatrix 的矩阵统一以 mat4 类型来表示，通过工厂函数 mat4.create() 来创建矩阵对象。函数 initMatrices() 创建了模型-视图矩阵和投影矩阵，并用全局变量 modelViewMatrix 和 projectionMatrix 来存储这两个矩阵。

例 2-4：初始化投影矩阵和模型-视图矩阵

```
var projectionMatrix, modelViewMatrix;

function initMatrices(canvas)
{
    // 创建一个模型-视图矩阵,包含一个位于(0, 0, -3.333)的相机
    modelViewMatrix = mat4.create();
    mat4.translate(modelViewMatrix, modelViewMatrix, [0, 0, -3.333]);

    // 创建一个45度角视野的投影矩阵
    projectionMatrix = mat4.create();
    mat4.perspective(projectionMatrix, Math.PI / 4,
        canvas.width / canvas.height, 1, 10000);
}
```

注 5：在 3D 图形处理中，当相机的位置被移动时，程序实际进行的处理是根据当前定义的相机位置去对整个场景进行平移，例如相机位于 [0, 0, 3] 位置时，实际上会被处理为整个场景进行了 [0, 0, −3] 的平移。——译者注

2.4.5 着色器

场景绘制的准备工作已经差不多了。接下来我们要进行至关重要的初始化环节：着色器。正如先前所提到过的，着色器是一段使用 GLSL（一种类 C 的高级语言）编写的简短程序，它定义了 3D 对象的像素点实际绘制到屏幕上的方式。WebGL 要求开发者为每个待绘制的对象提供一个着色器。一个着色器可以应用于多个对象，因此在实际应用中，整个场景通常只需提供一个统一的着色器。通过设置不同的参数，可以在不同的几何形状上复用它。

一个着色器通常由两个部分组成：*顶点着色器*（vertex shader）和*片段着色器*（fragment shader，又称 pixel shader，像素着色器）。顶点着色器负责将物体的坐标转换为 2D 显示区域中的坐标；片段着色器负责计算转换好的顶点像素的最终颜色输出，其基于颜色、纹理、光照、材质等数值输入。我们简单示例中的顶点着色器包含顶点位置 `vertexPos`、模型 - 视图矩阵 `modelViewMatrix` 以及投影矩阵 `projectionMatrix` 的数值，这些数值与输入数值结合计算，构建出最终的、平移好的顶点数据，而片段着色器则简单地输出设定好的白色。

在 WebGL 中，初始化着色器需要进行一系列步骤，包括将独立的 GLSL 源代码片段编译到一起并建立链接。例 2-5 列出了着色器代码，让我们来通读这段代码。首先定义一个辅助函数 `createShader()`，这个函数调用 WebGL 提供的方法来编译顶点着色器和片段着色器的源代码。

例 2-5：着色器代码

```
        function createShader(gl, str, type) {
        var shader;
        if (type == "fragment") {
            shader = gl.createShader(gl.FRAGMENT_SHADER);
        } else if (type == "vertex") {
            shader = gl.createShader(gl.VERTEX_SHADER);
        } else {
            return null;
        }

        gl.shaderSource(shader, str);
        gl.compileShader(shader);

        if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
            alert(gl.getShaderInfoLog(shader));
            return null;
        }

        return shader;
    }
```

GLSL 源代码以 JavaScript 字符串的形式定义，存储在全局变量 `vertexShaderSource` 和 `fragmentShaderSource` 中：

```
        var vertexShaderSource =

            "    attribute vec3 vertexPos;\n" +
            "    uniform mat4 modelViewMatrix;\n" +
```

```
"    uniform mat4 projectionMatrix;\n" +
"    void main(void) {\n" +
"        // 返回经过投影和变换的顶点值\n" +
"        gl_Position = projectionMatrix * modelViewMatrix * \n" +
"            vec4(vertexPos, 1.0);\n" +
"    }\n";

var fragmentShaderSource =
"    void main(void) {\n" +
"    // 返回像素点的颜色:始终输出白色\n" +
"    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);\n" +
"}\n";
```

GLSL 代码由存储在全局变量中的 JavaScript 字符串提供。这有点讨厌，因为我们不得不用加号来连接不同行，来保证代码格式。作为替代方案，我们可以采用先在外部文本文件中定义着色器，然后用 Ajax 的方式来加载这个文件。或者我们也可以创建隐藏的 DOM 节点，然后把代码写在 DOM 节点的文本内容中。为了便于说明，我们在示例代码中使用了这种最简单的形式。当真正编写代码的时候，你可以选择其他更优雅的方式。

当着色器的各个部分被编译完成，我们需要调用 WebGL 中的 `gl.createProgram()`、`gl.attachShader()` 以及 `gl.linkProgram()` 方法将它们链接到同一段程序中。随后我们需要调用 `gl.getAttribLocation()` 和 `gl.getUniformLocation()` 函数获取 GLSL 程序中定义的各个变量的句柄，从而可以用 JavaScript 中定义的数值来初始化这些变量。[6] `initShader()` 函数的定义如下：

```
var shaderProgram, shaderVertexPositionAttribute,
    shaderProjectionMatrixUniform,
    shaderModelViewMatrixUniform;

function initShader(gl) {

    // 加载并编译片段和顶点着色器
    var fragmentShader = createShader(gl, fragmentShaderSource,
        "fragment");
    var vertexShader = createShader(gl, vertexShaderSource,
        "vertex");

    // 将它们链接到一段新的程序中
    shaderProgram = gl.createProgram();
    gl.attachShader(shaderProgram, vertexShader);
    gl.attachShader(shaderProgram, fragmentShader);
    gl.linkProgram(shaderProgram);

    // 获取指向着色器参数的指针
    shaderVertexPositionAttribute =
        gl.getAttribLocation(shaderProgram, "vertexPos");
```

注 6：这里描述的是将 GLSL 中的变量转换为 JavaScript 变量的过程，从而使得 JavaScript 可以向 GLSL 的变量注入一些配置参数。——译者注

```
        gl.enableVertexAttribArray(shaderVertexPositionAttribute);

        shaderProjectionMatrixUniform =
           gl.getUniformLocation(shaderProgram, "projectionMatrix");
        shaderModelViewMatrixUniform =
           gl.getUniformLocation(shaderProgram, "modelViewMatrix");


        if (!gl.getProgramParameter(shaderProgram,
            gl.LINK_STATUS)) {
            alert("Could not initialise shaders");
        }
    }
```

2.4.6 绘制图元

现在，我们已经为绘制正方形做好了全部的准备工作——我们创建了绘图上下文；设置了视口；顶点缓冲、矩阵和着色器也都已创建和初始化。下面我们定义一个函数 `draw()`，用它来绘制我们在上文中展示过的那个正方形。让我们来通读这个函数。

首先，`draw()` 函数以黑色背景填充的方式清空了整个画布，方法 `gl.clearColor()` 将黑色设为当前画布的“清空”颜色。这个方法携带四个参数，分别代表 RGBA（Red、Green、Blue、Alpha）颜色的四个分量。注意 WebGL 的 RGBA 值是用 0.0 到 1.0 范围的浮点数来表示的（这与用 0~255 的整数表示的 Web 颜色值不一样，例如在 CSS 中）。`gl.clear()` 使用定义的“清空”颜色来“清空”WebGL *颜色缓冲*（color buffer），即 GPU 显存中用于渲染屏幕上像素点的区域。[7]［WebGL 使用多种类型的*缓冲*（buffer）来进行绘制，包括颜色缓冲和用于深度测试的*深度缓冲*（depth buffer）。关于深度缓冲，我们将在下一节予以说明。］

其次，`draw()` 函数将正方形的顶点缓冲数据绑定到绘图上下文的缓冲，设定了图元绘制过程中将要使用的着色器，并建立顶点缓冲数据和矩阵与着色器之间的关联。

最后，我们调用 WebGL 的 `drawArrays()` 函数来绘制这个正方形。WebGL 通过传入的图元类型参数和图元的顶点数量参数，并结合之前预设的其他属性（顶点、矩阵、着色器）来得到最终的绘制结果。例 2-6 展示了整个流程。

例 2-6：绘制代码

```
function draw(gl, obj) {

    // 清空背景(使用黑色填充)
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT);

    // 设置待绘制的顶点缓冲
    gl.bindBuffer(gl.ARRAY_BUFFER, obj.buffer);

    // 设置待用的着色器
    gl.useProgram(shaderProgram);
```

注 7：指将颜色缓冲中的所有字节都设为指定的“清空”颜色。——译者注

```
    // 建立着色器参数之间的关联:顶点和投影/模型矩阵
    gl.vertexAttribPointer(shaderVertexPositionAttribute,
        obj.vertSize, gl.FLOAT, false, 0, 0);
    gl.uniformMatrix4fv(shaderProjectionMatrixUniform, false,
        projectionMatrix);
    gl.uniformMatrix4fv(shaderModelViewMatrixUniform, false,
        modelViewMatrix);

    // 绘制物体
    gl.drawArrays(obj.primtype, 0, obj.nVerts);
}
```

终于，我们完成了整个绘制流程。程序执行的结果是一个绘制在黑色背景上的白色正方形，如之前图 2-1 所示的那样。

2.5 创建3D几何体

上面绘制的正方形是一个尽可能简单的 WebGL 示例。显然它不怎么能引起你的兴趣，甚至还不是 3D 的——尽管为了绘制这个正方形，我们已经编写了将近 200 行代码。而实现同样效果的 2D Canvas 绘制代码顶多只需 30 行左右。在这一点上，WebGL 相对其他绘图 API 没有显示出优势。但别着急，现在我们来用 WebGL 做一些有趣的事情——真正的 3D 绘图。为了得到一个包含不同颜色的 3D 立方体，我们需要在正方形的基础上加一些线条，为此我们将对着色器和绘图函数做一些小改动。我们还要为这个立方体加上一个简单的动画，以便从各个角度去观察它。图 2-2 展示了一个旋转中立方体的屏幕截图。

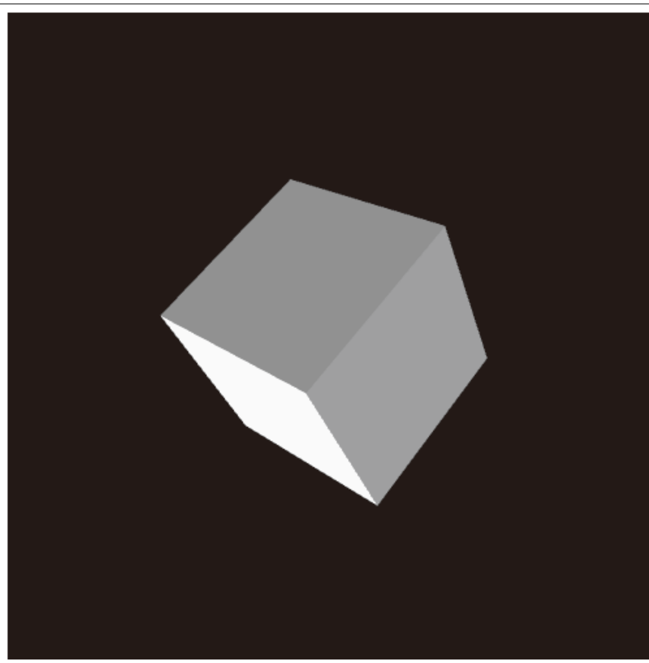


图 2-2：一个包含不同颜色的立方体

为了创建和渲染这个立方体，我们将改进之前的示例代码。首先，我们将用于创建正方形的缓冲数据改为用于创建立方体的缓冲数据。其次，我们将使用与之前不同的 WebGL 函数来执行绘制过程。文件 Chapter 2/example2-2.html 中包含绘制立方体的完整代码。

例 2-7 展示了立方体的缓冲设置过程，它比绘制正方形的代码要复杂一些。立方体有更多的顶点，并且我们将为不同的面设置不同的颜色。首先，我们创建顶点缓冲数据，并将它存储在变量 `vertexBuffer` 中。

例 2-7：初始化立方体、颜色和索引缓冲的代码

```
// 为彩色的立方体构建顶点、颜色和索引数据
function createCube(gl) {

    // 顶点数据
    var vertexBuffer;
    vertexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
    var verts = [
       // 正面
       -1.0, -1.0,  1.0,
        1.0, -1.0,  1.0,
        1.0,  1.0,  1.0,
       -1.0,  1.0,  1.0,

       // 背面
       -1.0, -1.0, -1.0,
       -1.0,  1.0, -1.0,
        1.0,  1.0, -1.0,
        1.0, -1.0, -1.0,

       // 顶面
       -1.0,  1.0, -1.0,
       -1.0,  1.0,  1.0,
        1.0,  1.0,  1.0,
        1.0,  1.0, -1.0,

       // 底面
       -1.0, -1.0, -1.0,
        1.0, -1.0, -1.0,
        1.0, -1.0,  1.0,
       -1.0, -1.0,  1.0,

       // 右面
        1.0, -1.0, -1.0,
        1.0,  1.0, -1.0,
        1.0,  1.0,  1.0,
        1.0, -1.0,  1.0,

       // 左面
       -1.0, -1.0, -1.0,
       -1.0, -1.0,  1.0,
       -1.0,  1.0,  1.0,
       -1.0,  1.0, -1.0
       ];
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(verts), gl.STATIC_DRAW);
```

其次，创建颜色数据，为每个顶点设置一个四元色，并将其存储在变量 `colorBuffer` 中。`faceColors` 数组中是一系列定义好的 RGBA 颜色值。

```
// 颜色数据
var colorBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
var faceColors = [
    [1.0, 0.0, 0.0, 1.0], // 正面
    [0.0, 1.0, 0.0, 1.0], // 背面
    [0.0, 0.0, 1.0, 1.0], // 顶面
    [1.0, 1.0, 0.0, 1.0], // 底面
    [1.0, 0.0, 1.0, 1.0], // 右面
    [0.0, 1.0, 1.0, 1.0]  // 左面
];
var vertexColors = [];
for (var i in faceColors) {
    var color = faceColors[i];
    for (var j=0; j < 4; j++) {
        vertexColors = vertexColors.concat(color);
    }
}
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertexColors),
    gl.STATIC_DRAW);
```

最后，我们要创建一类新型的缓冲——*索引缓冲*（index buffer），用于存储顶点数据的索引。我们将这些数据存储在变量 `cubeIndexBuffer` 中。之所以这里样做，是因为我们将在更新过的 `draw()` 函数中使用顶点集索引而非顶点本身来定义所有的三角形。这样做的理由是：3D 几何图形往往代表了连续封闭的区域，单个顶点常常由多个三角形共享，而索引缓冲能够避免数据重复，令数据存储更加紧凑。

```
// 索引数据(定义待绘制的三角形)
var cubeIndexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, cubeIndexBuffer);
var cubeIndices = [
    0, 1, 2,      0, 2, 3,    // 正面
    4, 5, 6,      4, 6, 7,    // 背面
    8, 9, 10,     8, 10, 11,  // 顶面
    12, 13, 14,   12, 14, 15, // 底面
    16, 17, 18,   16, 18, 19, // 右面
    20, 21, 22,   20, 22, 23  // 左面
];
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(cubeIndices),
    gl.STATIC_DRAW);

var cube = {buffer:vertexBuffer, colorBuffer:colorBuffer,
    indices:cubeIndexBuffer,
        vertSize:3, nVerts:24, colorSize:4, nColors: 24, nIndices:36,
        primtype:gl.TRIANGLES};
return cube;
}
```

为了绘制立方体的颜色，这些颜色必须被传递给着色器。例 2-8 展示了改进后的着色器代码。注意加粗的代码行：我们声明了一个代表顶点颜色的属性。此外我们还需要声明一个

GLSL 中的 varying 变量——vColor，它用于将每个顶点的颜色信息从顶点着色器传递到片段着色器。与之前出现过的并不逐个更改顶点数据的 uniform 变量（例如我们早先讨论的矩阵）不同，varying 变量代表着色器会为每个顶点逐个输出不同的值。在这个示例中，我们将存储在 vertexColor 变量中的颜色缓冲数据输入到变量 vColor 中。片段着色器直接输出 vColor 中的原始颜色值。

例 2-8：用于渲染带颜色正方体的着色器代码

```
    var vertexShaderSource =

        "    attribute vec3 vertexPos;\n" +
        "    attribute vec4 vertexColor;\n" +
        "    uniform mat4 modelViewMatrix;\n" +
        "    uniform mat4 projectionMatrix;\n" +
        "    varying vec4 vColor;\n" +
        "    void main(void) {\n" +
        "        // 返回经过变换和投影的顶点值\n" +
        "        gl_Position = projectionMatrix * modelViewMatrix * \n" +
        "            vec4(vertexPos, 1.0);\n" +
        "        // Output the vertexColor in vColor\n" +
        "        vColor = vertexColor;\n" +
        "    }\n";

    var fragmentShaderSource =
        "    precision mediump float;\n" +
        "    varying vec4 vColor;\n" +
        "    void main(void) {\n" +
        "    // 返回像素点颜色:始终输出白色\n" +
        "    gl_FragColor = vColor;\n" +
        "}\n";
```

如果仅仅用于设置单一的颜色，这段代码看起来也许有点过于复杂。但一个复杂的着色器——例如一个实现了光照模型的着色器，或者一个实现了草地、水面动态纹理的着色器，等等——在输出最终色彩之前，会对 vColor 进行许多额外的运算处理。无疑，着色器提供了强大的视觉能力，但是正如 Ben Parker 的名言所述——能力越大，责任越大。

现在我们开始编写用于绘制的代码，如例 2-9 所示。为了绘制比正方形更为复杂的立方体，我们需要做一些不同的事。示例代码中加粗的部分标明了这些改动。首先，我们要开启深度测试，使得 WebGL 可以按深度排序来绘制 3D 物体。否则，WebGL 将无法保证将“在前方”的面按照我们的预期绘制在其他面的前方，“前方”和“后方”的面会混淆在一起。（如果想看看关闭深度测试会发生什么事，只需注释掉那行代码。你仍然会看到立方体的部分面，但不完整。）

其次，我们要将之前已经在 createCube() 函数中创建好的颜色和索引缓冲绑定到绘图上下文的缓冲。最后，我们调用 WebGL 方法 gl.drawElements() 而不是 gl.drawArray() 来绘制用索引缓冲来表示的图元信息。

例 2-9：更改后的立方体绘制代码

```
function draw(gl, obj) {

    // 清空背景(使用黑色填充)
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.enable(gl.DEPTH_TEST);
    gl.clear(gl.COLOR_BUFFER_BIT  | gl.DEPTH_BUFFER_BIT);

    // 设置待用的着色器
    gl.useProgram(shaderProgram);

     // 建立着色器参数之间的关联:顶点和投影/模型矩阵
     // 设置缓冲
    gl.bindBuffer(gl.ARRAY_BUFFER, obj.buffer);
    gl.vertexAttribPointer(shaderVertexPositionAttribute,
        obj.vertSize, gl.FLOAT, false, 0, 0);
    gl.bindBuffer(gl.ARRAY_BUFFER, obj.colorBuffer);
    gl.vertexAttribPointer(shaderVertexColorAttribute,
        obj.colorSize, gl.FLOAT, false, 0, 0);
    gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, obj.indices);

    gl.uniformMatrix4fv(shaderProjectionMatrixUniform, false,
        projectionMatrix);
    gl.uniformMatrix4fv(shaderModelViewMatrixUniform, false,
        modelViewMatrix);

    // 绘制物体
    gl.drawElements(obj.primtype, obj.nIndices, gl.UNSIGNED_SHORT, 0);
}
```

2.6 添加动画

如果希望看到立方体的 3D 效果而不是一个静止的 2D 图像，我们需要让它动起来。现在，我们来为这个立方体添加一个绕坐标轴旋转的简单动画。动画的代码如例 2-10 所示。在函数 `animate()` 中，立方体以五秒钟为周期围绕预先定义的旋转轴 `rotationAxis` 旋转。

`animate()` 由另一个函数 `run()` 循环调用，它调用一个新的浏览器函数 `requestAnimationFrame()` 来持续驱动动画。这个函数在每次浏览器重绘页面的时候调取一个回调函数。（关于这个函数以及其他动画相关的技术，后续的章节里有更详细的说明。）每次 `animate()` 函数被调用的时候，它都会计算当前时间和上一次调用该函数的时间之间的差值，并将其存储在变量 `deltat` 中，然后根据它计算出作用于矩阵变量 `modelViewMatrix` 的旋转角度。代码执行的结果是立方体以每五秒一圈的速度绕 `rotationAxis` 旋转。

例 2-10：为立方体添加动画

```
var duration = 5000; // 毫秒(ms)
var currentTime = Date.now();
function animate() {
    var now = Date.now();
    var deltat = now - currentTime;
```

```
        currentTime = now;
        var fract = deltat / duration;
        var angle = Math.PI * 2 * fract;
        mat4.rotate(modelViewMatrix, modelViewMatrix, angle, rotationAxis);
    }

    function run(gl, cube) {

        requestAnimationFrame(function() { run(gl, cube); });
        draw(gl, cube);
        animate();
    }
```

2.7 使用纹理映射

纹理映射是我们在本章要学习的最后一个 WebGL API 特性。纹理映射（texture map），或简称纹理，是指覆盖几何体表面显示的位图。WebGL 中使用 `Image` DOM 元素作为纹理数据的源，这表示，只需简单地更改 `Image` 元素的 `src` 属性，你就可以将 Web 图像格式，例如 JPEG 和 PNG，作为 WebGL 贴图了。

WebGL 纹理并不一定要以图像文件为源来构建。2D canvas 元素也可以作为纹理的源，利用这个特性，我们可以使用 2D Canvas 绘图 API 在 3D 物体的表面绘制图案；纹理甚至还可以以 `Vedio` 元素作为源来构建，因此你可以在一个 3D 物体的表面播放视频。关于动态纹理的能力，在第 11 章有更详细的说明。

我们将前面的旋转立方体示例改为使用纹理映射而非表面色彩。如图 2-3 所示。



图 2-3：一个使用了纹理映射的立方体

这里需要提醒各位读者，如果你直接在文件系统里双击打开纹理映射的代码示例页面，它是无法正常运行的。它需要用一个 Web 服务器来加载，因为我们从一个 JPEG 文件中载入纹理，这是由于 WebGL 安全模型中的跨域访问安全限制，我们需要运行一个 Web 服务器而非直接通过 file:// URL 来访问这个文件。总的来说，本书中的大多数例子都需要通过 Web 服务器来访问。

我在 MacBook 上运行了一个标准本地版 LAMP 环境，不过你需要用到的仅仅是 LAMP 的一部分功能——像 Apache 这样的 Web 服务。或者如果你的机器上装了 Python，你也可以利用 Python 内置的 SimpleHTTPServer 模块来启动一个 Web 服务，使用命令行窗口定位到 examples 目录，然后输入：

```
python -m SimpleHTTPServer
```

这样你就可以通过 http://localhost:8000/ 这个地址来访问本书的示例了。如果希望获取更多关于这方面的技术支持，请访问 Linux Journal 网站（http://bit.ly/linuxjournal-http-python）。

这个示例的完整代码在文件 Chapter 2/example2-3.html 中。例 2-11 展示了加载纹理的代码。首先，我们调用 `gl.createTexture()` 来创建一个新的 WebGL 纹理对象。然后将这个纹理对象的 `image` 属性设为一个新创建的 `Image` 对象。最后，将这个 `Image` 对象的 `src` 属性设为一个 JPEG 文件的路径——在这个示例中，我们加载了一个 256 × 256 的正方形 WebGL 官方 LOGO——不过首先我们要为图像的 `onload` 事件注册一个事件处理程序，以便在图像加载完毕的时候对 WebGL 纹理对象做一些处理。

例 2-11：从图像创建一个纹理映射

```
var okToRun = false;

function handleTextureLoaded(gl, texture) {
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
        texture.image);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
    gl.bindTexture(gl.TEXTURE_2D, null);
    okToRun = true;
}

var webGLTexture;

function initTexture(gl) {
    webGLTexture = gl.createTexture();
    webGLTexture.image = new Image();
    webGLTexture.image.onload = function () {
        handleTextureLoaded(gl, webGLTexture)
    }

    webGLTexture.image.src = "../images/webgl-logo-256.jpg";
}
```

在 `onload` 事件的回调函数 `handleTextureLoaded()` 中，我们做了下面这些事情。首先，我们调用 `gl.bindTexture()` 函数来指定 WebGL 在后续绘制过程中将要使用的纹理。被指定的纹理将在后续的整个绘制过程中生效，直到 `gl.bindTexture()` 再次被调用——在函数的末尾，我们将绑定纹理设置为空，以防止在后续操作中意外更改纹理存储区域的内容。

其次，我们调用 `gl.pixelStorei()` 函数来翻转所有纹理像素点的 *y* 坐标值。之所以需要进行这个操作，是因为在 WebGL 中，纹理坐标系的 *y* 轴是垂直向上的，而在 Web 图像本身的坐标系中，*y* 轴是垂直向下的。

> `gl.pixelStorei()` 函数名中的字母 i 代表整型数（integer），WebGL 的函数命名规范参照 OpenGL，通常以一个字母后缀来标识函数参数的类型。图像以整型数组的方式存储（RGB 或 RGBA 颜色），因此被标识为 i。

现在我们调用 `texImage2D()` 方法来将加载好的图像数据复制到 WebGL 纹理对象中。这个方法支持几种不同类型的参数，你可以查阅 WebGL 规范了解如何使用它创建不同类型的纹理。在这个示例中，我们在第 0 层创建了一个 2D 纹理——一个纹理中可以包含不同层级的纹理，这个技术被称为 mip-mapping，我们将在后面进行介绍——这个纹理采用了 RGBA 颜色模式，存储在一个无符号字节（unsigned byte）数组中。

我们还需要设置纹理过滤选项，这些选项用于控制图像随远近位置放大缩小时的纹理像素颜色计算。在我们的示例中，我们使用了最简单的过滤设置——`gl.NEAREST`，这个设置的策略是通过缩放图像本身来得出纹理像素点的颜色。在这个设置下，纹理在没有被过度缩放的前提下看起来效果还不错，但过近处（放大）会呈现块状和像素化效果，而过远处（缩小）会呈现锯齿和不平滑效果。WebGL 提供了另外两种纹理过滤能力：`gl.LINEAR` 是使用线性插值的方法来处理放大，使得放大后纹理的视觉效果更加平滑；`gl.LINEAR_MIPMAP_NEAREST` 用于添加 mip-map 过滤，使得远处物体的纹理看起来更平滑。

想要感受 `gl.NEAREST` 过滤的缺点，可以尝试调整立方体的位置。修改源文件 Chapter 2/example2-3.html 的第 47 行，修改立方体 *z* 坐标的值（当前为 -8），调整立方体的远近。

```
mat4.translate(modelViewMatrix, modelViewMatrix, [0, 0, -8]);
```

尝试将 −8 改为 −4。当立方体更加靠近观察点，你可以观察到立方体的纹理明显变得像素化了（图 2-4）。

现在再尝试将 −8 改成 −32。当立方体远离观察点时候，可以看到纹理出现了明显的锯齿（图 2-5）。



图 2-4：gl.NEAREST 过滤——近处物体上的纹理变得像素化

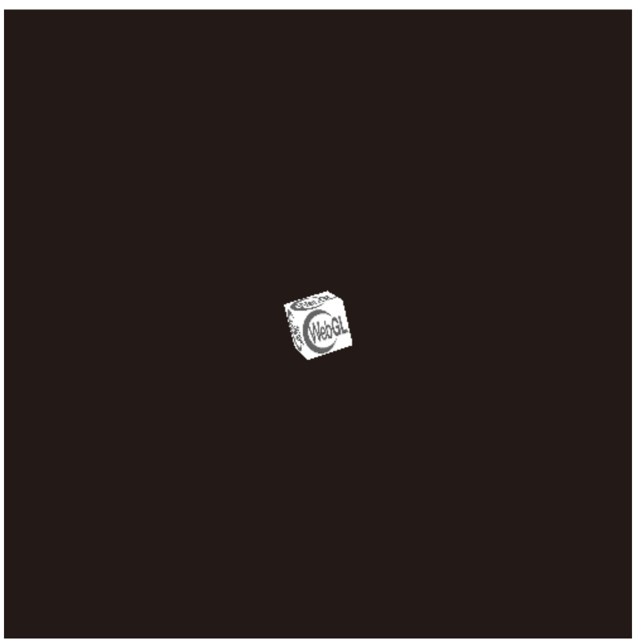


图 2-5：gl.NEAREST 过滤——较远物体的纹理出现锯齿

现在，我们已经设置好了全部的纹理选项，并调用 `gl.bindTexture()` 方法来清空当前绑定的纹理。最后，我们将全局变量 `okToRun` 的值设为 `true`，以此通知 `run()` 函数纹理已经准备就绪，绘图程序可以开始运行了。

像往常一样，我们还需要对代码的其他部分进行更改：缓冲的创建代码，着色器代码，以及着色值的设置代码。首先，我们将创建颜色缓冲的代码替换为创建纹理缓冲的代码。纹理坐标（texture coordinate）用随顶点数据一同定义的、[0, 1] 范围内的一对浮点数来表示。这对浮点数对应位图上的 *x*、*y* 偏移量，正如下面着色器的代码中所展现的那样。对我们的立方体来说，纹理坐标值非常简单：我们把整张纹理分别贴到立方体的每个表面上，因此立方体每个面的转角坐标恰好对应纹理图像的转角坐标，如 [0, 0]、[0, 1]、[1, 0] 或 [1, 1]。注意，这些纹理坐标值的顺序与顶点缓冲中的顶点顺序是相对应的。例 2-12 展示了创建纹理坐标缓冲的代码。

例 2-12：纹理映射立方体的缓冲创建代码

```
var texCoordBuffer = gl.createBuffer();
 gl.bindBuffer(gl.ARRAY_BUFFER, texCoordBuffer);

 var textureCoords = [
   // 正面
   0.0, 0.0,
   1.0, 0.0,
   1.0, 1.0,
   0.0, 1.0,

   // 背面
   1.0, 0.0,
   1.0, 1.0,
   0.0, 1.0,
   0.0, 0.0,

   // 顶面
   0.0, 1.0,
   0.0, 0.0,
   1.0, 0.0,
   1.0, 1.0,

   // 底面
   1.0, 1.0,
   0.0, 1.0,
   0.0, 0.0,
   1.0, 0.0,

   // 右面
   1.0, 0.0,
   1.0, 1.0,
   0.0, 1.0,
   0.0, 0.0,

   // 左面
   0.0, 0.0,
   1.0, 0.0,
   1.0, 1.0,
```

```
    0.0, 1.0,
  ];
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(textureCoords),
      gl.STATIC_DRAW);
```

我们还要将使用色彩的着色器更改为使用纹理的着色器。在顶点着色器中，我们定义了一个名为 texCoord 的顶点属性变量，顶点数据通过这个变量传入，以及一个 varying 类型的输出变量 vTexCoord，用于向片段着色器输出各个顶点的信息。片段着色器使用纹理坐标作为纹理映射数据的索引，纹理坐标通过 uniform 变量 uSampler 传入程序。我们调用 GLSL 函数 texture2D() 从纹理中取得像素信息，包括采样器和以 (*x*, *y*) 形式存储的 2D 顶点位置。更改后的着色器代码如例 2-13 所示。

例 2-13：纹理映射立方体的着色器代码

```
var vertexShaderSource =

    "    attribute vec3 vertexPos;\n" +
    "    attribute vec2 texCoord;\n" +
    "    uniform mat4 modelViewMatrix;\n" +
    "    uniform mat4 projectionMatrix;\n" +
    "    varying vec2 vTexCoord;\n" +
    "    void main(void) {\n" +
    "        // 返回经过变换和投影的顶点值\n" +
    "        gl_Position = projectionMatrix * modelViewMatrix * \n" +
    "            vec4(vertexPos, 1.0);\n" +
    "        // Output the texture coordinate in vTexCoord\n" +
    "        vTexCoord = texCoord;\n" +
    "    }\n";

var fragmentShaderSource =
    "    precision mediump float;\n" +
    "    varying vec2 vTexCoord;\n" +
    "    uniform sampler2D uSampler;\n" +
    "    void main(void) {\n" +
    "    // 返回像素点的颜色:始终输出白色\n" +
    "    gl_FragColor = texture2D(uSampler, vec2(vTexCoord.s, vTexCoord.t));\n" +
    "}\n";
```

为了将纹理贴到我们的立方体上，我们最后还要对绘制函数做一些小修改。例 2-14 展示了修改后的代码。我们将设置颜色缓冲的代码替换为设置纹理缓冲的代码，并将该纹理设为当前纹理，绑定到绘图上下文。

例 2-14：初始化绘制所需的纹理映射数据

```
  gl.vertexAttribPointer(shaderTexCoordAttribute, obj.texCoordSize, gl.FLOAT,
      false, 0, 0);
  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, obj.indices);

  gl.uniformMatrix4fv(shaderProjectionMatrixUniform, false, projectionMatrix);
  gl.uniformMatrix4fv(shaderModelViewMatrixUniform, false, modelViewMatrix);

  gl.activeTexture(gl.TEXTURE0);
  gl.bindTexture(gl.TEXTURE_2D, webGLTexture);
  gl.uniform1i(shaderSamplerUniform, 0);
```

2.8 小结

本章讲述了如何使用 WebGL API 进行图形渲染。我们了解了编写一个 WebGL 应用的基本流程，包括创建绘图上下文、视口、缓冲、矩阵、着色器和图元的绘制。我们学习了如何创建 2D 和 3D 几何图形并用颜色和纹理来填充它们。我们还稍微借助了开源库 glMatrix 和 RequestAnimationFrame.js，它们都是 WebGL 开发的常用基本库。

显然，到目前为止我们接触到的 WebGL 底层编程是非常繁琐的。在读完本章之后，我们已经可以编写一些稍微复杂的、包括颜色和纹理的几何图形，尽管为此可能需要编写数百行代码。这提供了强大的能力——你可以精细地操作屏幕上的每一个顶点和像素，以令人炫目的硬件加速的速度进行。然而使用 WebGL 底层接口来编写代码需要大量繁重的工作。标准的设计者们采用了牺牲代码尺寸来换取更强大能力的思路。WebGL 的 API 小而简单，这就使得更多工作落在了应用开发者身上。

如果你是一名经验丰富的游戏或图形开发者，并且希望更好地掌控应用的性能和特性，那么直接使用 WebGL API 对你来说也许是最好的选择。如果你正在开发的应用对渲染有特别的需求，例如图像处理应用或 3D 建模工具，那么你应该深入地研究 WebGL 中的 metal 技术。又或许，你需要开发一些更顶层的应用，例如谁也不希望为了创建一个立方体而重复写那四十行相同的代码，在这个层次上你得完全靠自己，你需要理解和控制每一行代码。

尽管如此，如果你和大多数人一样对 3D 并不是特别熟悉，那么你应该以一种比 WebGL API 本身更高级的封装方式来编写应用，例如使用一些现成的工具。事实上这些工具已经存在：有许多基于 WebGL 的优秀开源库可供使用。我们将在后面的几章一一介绍它们。

第3章

Three.js——一款JavaScript3D引擎

我们在上一章展示了 WebGL 编程的强大和繁琐。WebGL 允许你使用 GPU 全部的能力来创建美丽的实时渲染、带动画的 Web 页面。然而如果你希望用这些最基础的 API 去做一些令人惊叹的事，那么你将需要付出大量的努力，并逐行编写所需的成百上千行代码。在这个以快为主的 Web 时代，这样的方式显然不适合用于构建应用。面对项目需求，大多数的开发者面临着两个选择：创建自己的辅助库来提升开发效率，或者使用现成的第三方库。

而可用于辅助展开 WebGL 开发的第三方库同样有很多，毋庸置疑，Three.js（http://threejs.org/）是它们中的领导者。Three.js 提供了一套简易且直观的创建 3D 图形中常见物体的方案。它使用了许多最佳实践的图形引擎技术，速度很快。它还内置了多种类型的对象和方便的工具，功能非常强大。Three.js 是托管在 GitHub 上的开源项目并且维护良好，许多作者都在向这个项目贡献代码。

Three.js 已经成为 WebGL 开发的事实选择。你在互联网上能看到的大多数 WebGL 优秀内容都是使用 Three.js 来构建的，包括 Google 的 100 000 Stars（见第 1 章），以及许多可以通过互联网访问到的富交互和高度创新的杰作。

3.1 使用Three.js创建的代表性项目

现今最广为人知的 WebGL 项目莫过于 RO.ME“3 Dreams of Black”（http://www.ro.me/），它是电影制作人 Chris Milk 于 2011 年在 Google 工程师的帮助下创建的一个交互式音乐作品。这是 Danger Mouse 和 Daniele Luppi 合作，并邀请 Jack White 和 Norah Jones 友情参与的专辑 ROME 中的歌曲“Black”的音乐影片，见图 3-1。



图 3-1：RO.ME“3 Dreams of Black”，由 ROME 专辑中的歌曲“Black”衍生的交互式音频体验

RO.ME 是一个完整的虚拟世界，用户可以自由地控制相机视角以及往场景里添加物体，同时也可以看到其他用户添加的物体。这个项目使用 Three.js 开发，它创造了在当时来说突破性的 WebGL 效果，包括用于使近处物体锐化、远处物体模糊的景深着色器，用于创造赛璐珞动画风格视觉效果的赛璐珞（toon）着色器，植绒行为，以及点云系统。如果希望了解更多 RO.ME 背后的技术，请访问团队的项目主页（http://www.ro.me/tech/）。

从梦幻电影到仿真写实，尽管效果完全不同，但 Three.js 在这些 WebGL 项目的构建中起到了同等重要的作用。借助 Three.js，人们可以创造出产品级的可视化效果。图 3-2 中的获奖汽车配置案例由德国团队 Plus 360 Degrees 创建，它允许用户旋转场景，从一系列精良的汽车模型中选择自己喜欢的模型，并自由改变其涂装颜色和轮胎，从而创造出个性化的汽车。类似的汽车配置应用多年前已经存在，甚至还有使用 flash 开发、运行在浏览器中的版本；尽管如此，这个演示的产品价值还是远远高于过去我们在互联网上看到的那些。精致的汽车细节模型，使用环境贴图来模拟反射，加上光照和阴影，组合成了高度仿真的视觉效果——这的确是个非常令人惊叹的交互式应用。

Three.js 不仅擅长渲染实物，在展示抽象概念方面也丝毫不弱。图 3-3 展示了这方面的一个惊人案例，一个作为 Google 实验项目创建的全球小型武器交易数据可视化项目。这个名为 Small Arms Imports/Exports 的项目展示了超过 100 万个分别代表小型武器进口和出口情况的数据点，并使用颜色、线条和发光效果在一个地球模型上连接和标注它们，用以表明 1992 到 2010 年间小型武器、轻武器以及弹药在 250 个国家和地区之间的交易情况。这些线条构成的网络清楚地展示了信息，并且视觉效果令人震撼。



图 3-2：汽车配置和可视化，作者是 Plus 360 Degrees



图 3-3：小型武器进出口情况，由 Google Ideas 创建的实验性项目（http://www.chromeexperiments.com/detail/arms-globe/）

Three.js 并不是一个传统意义上的游戏引擎（稍后详述）；尽管如此，它还是可以作为游戏引擎的底层支持。我们可以在它之上构建游戏引擎，并在此基础上开发出不错的游戏。为了向 Wipeout 和 F-Zero 系列游戏致敬，Thibaut Despoulain 开发了 HexGL——一个未来太空背景的竞速游戏。HexGL 具有很高的产品价值，它包括发光效果、粒子系统、对建筑物和船只的真实感渲染、利用后渲染技术构建的运动模糊线，以及完美的整体显示效果。图 3-4 展示了 HexGL 的实际效果。

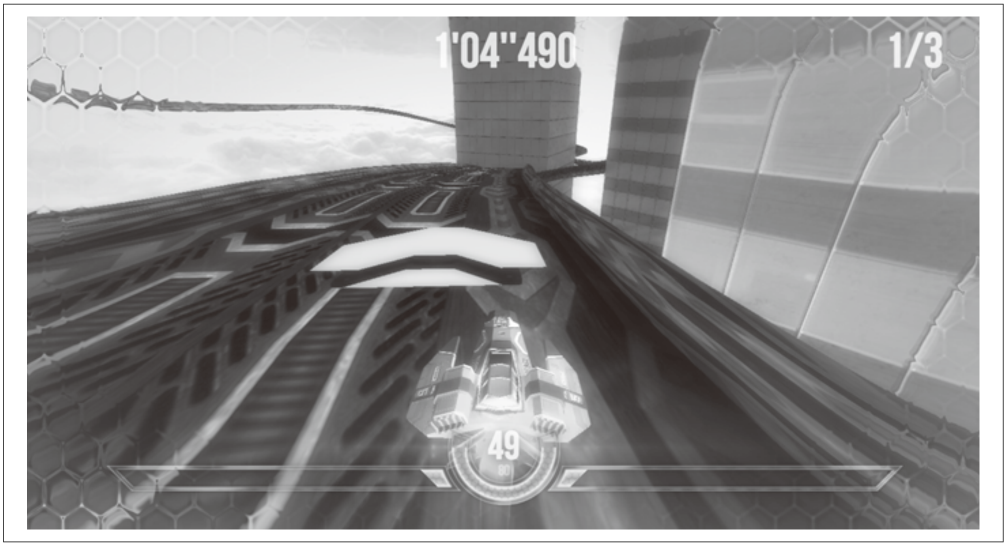


图 3-4：HexGL，一个未来感、快节奏的竞速游戏，由 Thibaut Despoulain 使用 HTML5、JavaScript 和 WebGL 创建

3.2 Three.js概览

Three.js 的作者是巴塞罗那的 Ricardo Cabello Miguel，或许你更熟悉他的另一个称呼 Mr.doob（我没敢问这是为什么）。Three.js 由 Mr.doob 早期（近十年前）在 3D 作品开发展示会（今天人们称之为黑客马拉松）上的作品发展而来。在现有工具都不能满足需求的情况下，Mr.doob 开始开发自己的工具，这套工具一开始是用运行在 Adobe Flash 平台上的 ActionScript 编写的。几年后 Google Chrome、高效的 JavaScript 以及 HTML5 一一登场，Mr.doob 便将他的研究转移到了浏览器这个新的平台上。之后在 2010 年，Three.js 诞生了。Three.js 的第一个版本使用 SVG 和 2D Canvas 渲染。在此几个月后 WebGL 发布了，Three.js 便开始移植到 WebGL 上进行后续开发。这真是个壮举，但 Mr.doob 对此表示"并不难实现"，也许是因为他已经基于 WebGL 编写了其他两个渲染器吧。从那时开始，Three.js 逐渐强大和完善，并最终成为创建 WebGL 3D 应用的最流行选择。

我选择基于 Three.js 来编写本书中的示例，并不仅仅是由于它非常流行的缘故——尽管这也是一部分原因。首先，我在项目开发中使用了它，并且非常喜欢。其次，从功能的角度来说，我认为它是目前最完善的 WebGL 库。再次，它有许多核心的代码贡献者，这保证了它始终能够和实际的项目接轨。最后，它非常容易上手——事实上，易用性是它最大的优点。不过有一点请记住，Three.js 只是众多 WebGL 库中的一个，你还有其他的选择；如果你的项目需要，或者你愿意的话，动手编写一个自己的库也未尝不可。在本书中你将会逐步详细地了解 Three.js，现在我们先给出一个大纲。

- **Three.js 隐藏了 WebGL 渲染中的底层细节。** Three.js 简化了 WebGL API 的细节，将 3D 场景表示为网格、材质、灯光（即图形开发者常用的各种对象类型）。

- **Three.js 非常强大**。Three.js 不仅仅是 WebGL 的一个封装，它内置了许多可用于游戏开发、动画、演示、数据可视化、建模工具应用的非常有用的模型对象，还提供了用于特殊效果的后渲染机制。除了它本身的能力，Three.js 还拥有丰富的示例，你可以在你的项目中使用这些示例的代码。
- **Three.js 非常易用**。Three.js 的 API 基于友好和易学的理念设计。随库提供的许多示例能够帮助你更好地入门。
- **Three.js 运行速度很快**。Three.js 采用了 3D 图形的最佳实践，既保证了高性能，又不因此牺牲可用性。
- **Three.js 非常稳定**。它包含完备的错误检查、异常和控制台警告，让开发者可以准确地跟踪程序问题。
- **Three.js 支持交互**。WebGL 不具备原生的选中支持，这意味着你无法获取到鼠标经过了当前场景中的哪个物体。Three.js 提供了对选中的支持，使得为应用添加交互变得更加简单。
- **Three.js 支持数学计算**。Three.js 包含强大易用的 3D 数学运算对象，如矩阵、映射和向量。
- **Three.js 内置第三方文件格式支持**。你可以利用 Three.js 提供的接口载入以文本格式存储的、用流行的建模工具导出的 3D 模型。Three.js 也定义了自己专属的 JSON 和二进制 3D 模型文件格式。
- **Three.js 是面向对象的**。开发者基于设计优良的 JavaScript 对象编写代码，而不是仅仅去调用一些函数。
- **Three.js 是可扩展的**。无论你希望为 Three.js 添加一些特性，还是定义自己的个性化 Three.js，都是非常简单的事。如果现有的数据格式支持无法满足你的需求，那么你可以为特定的数据格式编写相应的插件。
- **Three.js 包含 2D Canvas、SVG、CSS 的渲染引擎**。尽管 WebGL 已经非常流行，但它并未被所有的浏览器支持，这表示对某些应用来说，WebGL 并不是最好的选择。幸好 Three.js 还可以使用 2D Canvas 和 SVG 元素来渲染大部分的内容。当运行环境不支持 3D Canvas 的时候，这为你的代码提供了一个优雅的降级方案。Three.js 仍然可以用来渲染和变换 CSS 元素，我们将在第 6 章对此进行详细说明。

需要提醒大家的是，Three.js 也有一些不擅长的事情。比方说，Three.js 并不是一个游戏引擎。它缺乏一些游戏引擎所需的常用特性，如公告牌、头像、有限状态机以及物理引擎。Three.js 也没有编写多人联机游戏所需的内置网络支持。你需要基于 Three.js 自行构建它，或是整合其他专用的代码库。同时 Three.js 也不是一个应用框架，它不具备一个框架应有的安装、卸载、事件处理和运行循环机制。在后续的章节中我们将逐步讲述如何使用开发框架来节省开发时间，并避免在每个项目中都重复实现一套相同的机制。最后，Three.js 也不是一个开发环境，在 Three.js 中并未包含用于创建完整 3D 应用的集成开发工具。

也就是说，除了以上这些 Three.js 不擅长的事以外，我们可以尽情来欣赏它的优点：一个为浏览器设计的、高性能、功能齐全、易用的 3D 渲染引擎。它非常强大。下面让我们来初步认识这个强大的引擎。

3.2.1 初始化Three.js

如果你希望使用 Three.js 来开发你的项目，首先需要从 GitHub 上获取它的最新代码包，项目地址是 https://github.com/mrdoob/three.js/。当你将整个代码仓库复制下来之后，可以在文件夹中找到 JavaSript 程序包的未压缩版本 build/three.js。（文件夹中还有程序包的压缩版本 build/three.min.js，你可以在项目完结发布的时候使用它。尽管如此，我还是建议你在研究本书示例的时候使用未压缩的版本，以便于调试。）同时也不要去改变 src 文件夹下的目录结构。尽管 Three.js 的 GitHub 仓库中同时包含了文档页面，但这仅仅是一些非常基础的说明，所以你需要保持原有的目录结构，以保证正确的引用。

> 本书中使用的 Three.js 版本是 revision 58（r58）。Mr.doob 及其团队一直频繁地更新版本，因此如果你下载的是最新版本的 Three.js，本书中的某些示例可能无法正确运行。本书中的全部示例代码都独立引用了一个 r58 版本的 Three.js 副本，该文件可以在目录 libs/three.js.r58/ 中找到。

3.2.2 Three.js工程结构

为了熟悉 Three.js，我们需要花一些时间来了解它的的目录结构、文档和示例。这里面包含的东西很多。你一定急着想要开始编写代码了，不过相信我，花一点时间来把这些东西看一遍，尤其是 example 目录下的内容。你不会后悔的。

下面我们来大致说明一下整个工程的文件目录结构。

- build/

  Three.js 的完整项目代码输出目录，包括未压缩版本和压缩版本。Three.js 使用 Google Closure 进行编译发布，编译好的输出文件包含完整的 Three.js 库代码，它由不同代码目录下的源代码编译而成。如果你不熟悉 Closure 并且希望了解更多关于它的信息，请访问 http://code.google.com/closure/compiler/。注意，你并不需要再次执行 Three.js 的编译发布过程。如果你并不想处理编译发布相关的事情，那么只需直接使用 build 目录中编译好的 three.js 或 three.min.js。

- docs/

  这个目录包含以 HTML 格式编写的完整 API 文档。虽然这份文档并不详细，但至少它为熟悉整个库提供了一份很好的设计概述。

- editor/

  Three.js 团队开发了一个用于创建 3D 场景的编辑系统。在本书写作时，这个工具尚处于开发期，还不能真正用于产品。但你应该信任 Mr.doob 的实力：因为只要给他一个浏览器和一个文本编辑器，就没有他不敢尝试的东西！

- examples/

  这个目录包含几百个示例。这些示例覆盖了 Three.js 的各种特性和效果，包括 Canvas、CSS 和 WebGL 渲染器的使用范例。其中一些是简单的“技术演示”，用于展现一些典型的特性；另一些则是综合运用了各类特性的精心大作。请花费一些时间来浏览每一个

示例以及阅读它们的代码，这是你熟悉并领略 Three.js 强大功能的最佳途径。

- src/

  整个库的代码文件目录。这是一个复杂的文件树结构，大致上可以分为两个部分：核心和扩展。核心部分包括主要的特性集合，这是 Three.js 可运行的基础。倘若这部分缺失，你将无法使用 Three.js 来渲染场景。扩展部分包含大量有用的特性，包括：内置的几何形状，如立方体、球体和圆柱；动画相关的通用函数；以及图片加载类。你完全可以基于 Three.js 核心自行构建这些东西，但你显然并不愿意这么做。因此，虽然属于扩展部分，但它们默认包含在编译输出的完整库文件中。

- utils/

  这个目录包含各种工具，包括用于编译压缩和非压缩版本库文件的 Google Closure 脚本，用于将各种 3D 文件格式转换为 Three.js JSON 和二进制文件格式（后续将会更加详细说明）的文件转换器，还提供了用于从建模软件导出 Three.js 格式文件的插件，如 Blender 和 Maya。

3.3 一个简单的Three.js程序

现在你对 Three.js 应当已经有了一个大致上的了解，是时候开始动手编写程序了。我们的第一个示例意在向大家清楚地展示这个库的价值——相对于在少得不能再少的 WebGL API 上开发而言。

回忆一下上一章的纹理映射立方体，在这里，我们将再一次实现它，但这次我们使用 Three.js。所需的 Three.js 代码如例 3-1 所示，完整的可运行代码可以在 Chapter 3/threejscube.html 文件中找到。

例 3-1：用 Three.js 创建一个纹理映射立方体

```
<script type="text/javascript">

    var renderer = null,
    scene = null,
    camera = null,
    cube = null;

    var duration = 5000; // ms
    var currentTime = Date.now();
    function animate() {

        var now = Date.now();
        var deltat = now - currentTime;
        currentTime = now;
        var fract = deltat / duration;
        var angle = Math.PI * 2 * fract;
        cube.rotation.y += angle;
    }

    function run() {
        requestAnimationFrame(function() { run(); });
```

```
            // 渲染场景
            renderer.render( scene, camera );

            // 为下一帧动画旋转立方体
            animate();

    }

    $(document).ready(
            function() {

                var canvas = document.getElementById("webglcanvas");

                // 创建Three.js渲染器并将其添加到canvas中
                renderer = new THREE.WebGLRenderer(
                  { canvas: canvas, antialias: true } );
                // 设置视口尺寸
                renderer.setSize(canvas.width, canvas.height);

                // 创建一个新的Three.js场景
                scene = new THREE.Scene();

                // 添加一个相机以便观察整个场景
                camera = new THREE.PerspectiveCamera( 45,
                  canvas.width / canvas.height, 1, 4000 );
                scene.add(camera);

                // 创建一个纹理映射的立方体并将其添加到场景中
                // 首先,创建纹理映射
                var mapUrl = "../images/webgl-logo-256.jpg";
                var map = THREE.ImageUtils.loadTexture(mapUrl);

                // 其次,创建一个基础材质,传入纹理映射
                var material = new THREE.MeshBasicMaterial({ map: map });

                // 接着,创建立方体几何形状
                var geometry = new THREE.CubeGeometry(2, 2, 2);

                // 其次,将几何形状和材质整合到一个网格中
                cube = new THREE.Mesh(geometry, material);

                // 将网格移动到与相机有一段距离的位置,并朝向观察者倾斜
                cube.position.z = -8;
                cube.rotation.x = Math.PI / 5;
                cube.rotation.y = Math.PI / 5;

                // 最后,将网格添加到场景中
                scene.add( cube );

                // 启动运行循环
                run();
            }
     );

</script>
```

这里的动画和运行循环代码与第 2 章中的代码十分相似，只有一些小的变化，稍后再解释。在这个版本中，比较显著的变化是立方体场景的创建：在之前使用了 WebGL 标准 API 的代码中，为实现这个需求，我们耗费了接近 300 行代码，然而在使用 Three.js 的示例中，我们仅仅需要 40 行代码。这里的 jQuery `ready()` 回调作用于整个页面。在这里更体现了这个特性。总的来说，这是一个普通的简单示例，但至少从这个示例中我们可以开始想象如何构建全面的应用，正如我们在本章开头所见过的那些一样。让我们来详细分析这个示例。

3.3.1 创建渲染器

首先，我们需要创建一个*渲染器*（render）。Three.js 使用插件式的渲染系统。我们可以使用不同的 API 来渲染相同的场景，例如对同一个场景，我们既可以使用 WebGL API 来渲染，也可以使用 2D Canvas API 来渲染。在这个示例中，我们创建了一个 `THREE.WebGLRenderer` 对象，并传入两个参数来对其进行初始化：`canvas`，正如字面上的意思，代表 HTML 文件中的 `<canvas>` 元素；`antialias` 标记，用于告知 Three.js 开启基于硬件的*多重采样抗锯齿*（multi-sample antialiasing，MSAA）策略。抗锯齿选项用于使渲染的物体边缘平滑，避免呈现锯齿。Three.js 使用这些参数来构建一个 WebGL 绘图上下文，并将这个绘图上下文添加到渲染器对象中。

创建好渲染器之后，我们将渲染器的尺寸设置为 canvas 对象的的宽和高，这与第 2 章中调用 `gl.viewport()` 来设置视口尺寸的操作是等效的。渲染器的全部设置只需两行代码：

```
// 创建Three.js渲染器并将其添加到canvas中
renderer = new THREE.WebGLRenderer(
  { canvas: canvas, antialias: true } );

// 设置视口尺寸
renderer.setSize(canvas.width, canvas.height);
```

3.3.2 创建场景

其次，我们通过创建一个 `THREE.Scene` 对象来创建一个*场景*（scene）。这个场景是 Three.js 图形层级结构的最顶层。它包含其他的全部图形对象。（在 Three.js 中，对象以父 – 子层级结构的形式存在。我们稍后会对此进行更详细的说明。）

现在我们有了一个场景，我们将为这个场景添加两个对象：一个*相机*（camera）和一个*网格*（mesh）。相机定义了我们观察场景的位置：在这个示例中，我们让相机停留在它的默认位置，即场景坐标系统的原点。在这里我们定义了一个 `THREE.PerspectiveCamera` 类型的相机，并用四个参数来初始化它。这四个参数分别是：45 度的视野角、宽高比、最近平面和最远平面的位置坐标值。在 Three.js 的内部，这些参数会被用来构建成一个透视投影矩阵，用于将 3D 场景渲染到 2D 绘图平面上。（如果你不记得相机、视口和投影是什么了，请翻回到第 1 章的图形基础内容。）

用于创建场景和添加相机的代码同样十分简洁：

```
// 创建一个新的Three.js场景
scene = new THREE.Scene();

// 添加一个相机以便观察整个场景
camera = new THREE.PerspectiveCamera( 45,
  canvas.width / canvas.height, 1, 4000 );
scene.add(camera);
```

现在是时候把网格添加到场景中了。一个网格包括一个几何形状（geometry）和一个材质（material）。我们使用 Three.js 内置的 `CubeGeometry` 对象来创建一个 2×2×2 的立方体。材质用于定义如何渲染物体的表面。在这个示例中我们定义了一个 `MeshBasicMaterial` 类型的材质，这是一个最简单的、不带光照效果处理的材质。我们将一张 WebGL logo 的图片作为纹理映射使用在立方体上。纹理映射，又称纹理，是用来表示 3D 网格模型表面属性的位图。纹理映射最简单的使用方法是定义一个表面颜色，而它们也可以通过组合来产生复杂的效果，例如凹凸或高光。

WebGL 提供了一系列纹理相关的 API，并提供了重要的安全特性，例如在纹理使用中对跨域访问的限制。令人兴奋的是，Three.js 提供了一个非常简单的 API，这个 API 可以用来加载纹理，并便利地将其和材质进行结合。我们调用 `THREE.ImageUtils.loadTexture()` 来从图片文件加载纹理，并通过材质构造时传入的 `map` 参数将得到的纹理和材质结合起来：

```
// 创建一个纹理映射的立方体并将其添加到场景中
// 首先,创建纹理映射
var mapUrl = "../images/webgl-logo-256.jpg";
var map = THREE.ImageUtils.loadTexture(mapUrl);

// 其次,创建一个基础材质,传入纹理映射参数
var material = new THREE.MeshBasicMaterial({ map: map });
```

在这里，Three.js 的底层代码做了一系列的事情。它将 JPEG 图片的像素点映射到立方体每个表面的正确位置上：图片既没有被拉伸覆盖整个立方体，每个面上的图片也没有颠倒或翻转。这看起来并没有什么大不了的。然而我们回顾前一章，如果用 WebGL 的原生方法，我们需要编写大量的代码来保证正确的映射，但使用了 Three.js，我们就只需短短的几行代码。

最后我们创建立方体的网格。到这里我们已经构建了几何形状、材质和纹理。我们将它们整合到一个变量名为 `cube` 的 `THREE.Mesh` 对象上。在将其添加到场景前，我们将立方体设置在相机后方八个单位长度的位置上，正如我们在第 2 章的示例中所做的那样，不同的是现在我们不必在面对那些繁琐的矩阵计算，只需简单地设置立方体的 `position.z` 属性。同时我们将立方体向观看者方向倾斜了一个角度，以便可以看清立方体的顶面，这只需设置立方体的 `rotation.x` 属性就可以做到。然后我们将这个立方体添加到场景中。好了！渲染场景所需的一切都准备就绪。

```
// 将网格移动到与相机有一段距离的位置,并朝向观察者倾斜
cube.position.z = -8;
cube.rotation.x = Math.PI / 5;
cube.rotation.y = Math.PI / 5;

// 最后,将网格添加到场景中
scene.add( cube );
```

3.3.3 运行循环的实现

和上一章的示例一样，我们需要使用 `requestAnimationFrame()` 函数来实现一个运行循环。不过实现的细节略有不同。在之前的版本里，我们的 `draw()` 函数需要设置缓冲、设置渲染状态、清空视口、设置着色器和纹理，等等。使用 Three.js，我们只需短短的一行代码：

```
renderer.render( scene, camera );
```

然后代码库会帮你处理其余的事情。在我看来，就这件事本身，已经足以成为选用 Three.js 的理由。

最后我们要展示的是将立方体旋转一个角度，以便能够更好地看清它的 3D 结构。有了 Three.js，这同样非常简单：只需将 `rotation.y` 属性设置为新的旋转角度，Three.js 的底层代码就会处理相关的矩阵运算，而不用我们自己动手去处理它。在下一次运行循环执行的时候，`render()` 函数会使用新的 y 旋转值，从而立方体会持续旋转。以下是 `animate()` 和 `render()` 函数的具体实现：

```
var duration = 5000; // ms
var currentTime = Date.now();
function animate() {

    var now = Date.now();
    var deltat = now - currentTime;
    currentTime = now;
    var fract = deltat / duration;
    var angle = Math.PI * 2 * fract;
    cube.rotation.y += angle;
}

function run() {
    requestAnimationFrame(function() { run(); });

        // 渲染场景
        renderer.render( scene, camera );

        // 为下一帧动画旋转立方体
        animate();
}
```

结果如图 3-5 所示，是不是很熟悉？

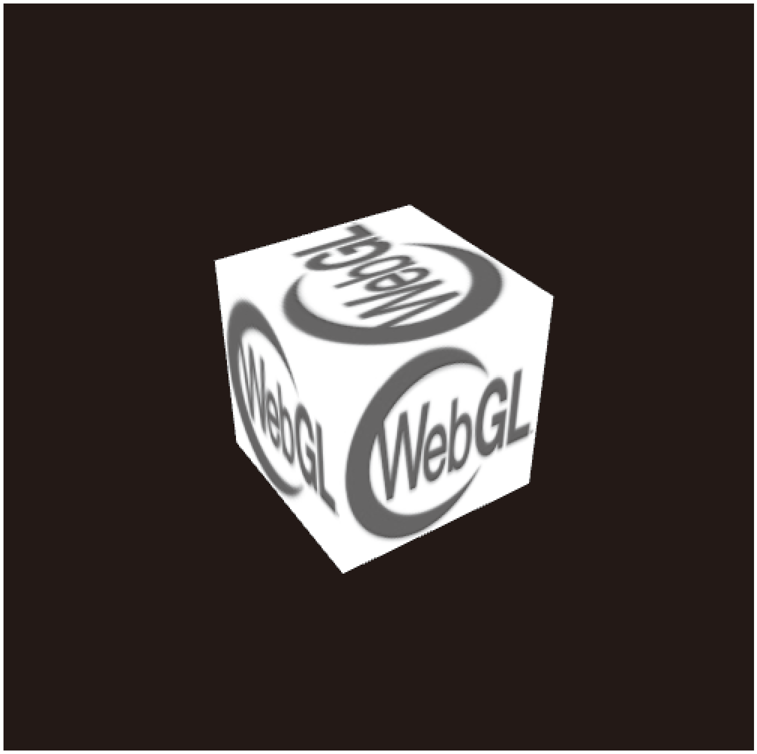


图 3-5：使用 Three.js 创建的纹理映射立方体

3.3.4 为场景添加光照

例 3-1 绘制了一个我们能够创建的最简单的 Three.js 3D 场景。但你可能已经注意到，在这个示例中，绘制出来的 3D 立方体看起来并不十分“3D”。当然，在立方体旋转的时候，我们通过各个面上的纹理可以粗略地看出立方体的形状。但整个场景中缺失了一个很重要的因素：明暗。实时 3D 渲染中令人惊叹的一点是：你可以使用灯光来打造一个物体表面的明部和暗部。如图 3-6 所示，现在立方体的各个面有了清晰的边缘，正如你在现实世界中看到的 3D 物体一样。我们通过向场景中添加灯光来实现这一点。

我曾经试图在第 2 章的例子中添加灯光。但为此我需要重写顶点和片段着色器，并编写大量的代码用于更新顶点缓冲数据，这显然并不值得；当时，我在斟酌是否应该浪费生命去编写古怪的 WebGL 代码，仅仅为了实现如此简单的事情。有了 Three.js，这简直不费吹灰之力就可以实现。我们仅仅需要新增几行代码。看看例 3-2 你就知道了。此版本的完整源代码可以在 Chapter 3/threejscubelit.html 中找到。



图 3-6：Three.js 立方体，带灯光，使用 Phong 着色方法

例 3-2：用 Three.js 为立方体添加光照

```
// 添加用于突出显示物体的定向光
var light = new THREE.DirectionalLight( 0xffffff, 1.5);

// 将灯光放置在场景外,指向原点
light.position.set(0, 0, 1);
scene.add( light );

// 创建一个带明暗的纹理映射立方体,并将其添加到场景中
// 首先,创建纹理映射
var mapUrl = "../images/webgl-logo-256.jpg";
var map = THREE.ImageUtils.loadTexture(mapUrl);

// 其次,创建一个Phong材质来展示明暗效果,传入纹理映射参数
var material = new THREE.MeshPhongMaterial({ map: map });
```

加粗显示的部分表明了整个过程。首先，我们往场景中添加一个灯光。灯光也是场景中的一类物体：当你创建它们并将它们添加到场景中时，它们的值将被用来渲染其他物体。在这个示例中，我们使用了一个*定向光*（directional light）。这是一束来自特定方向的平行光线。Three.js 用于构建定向光的语法（在我看来）有点违反直觉：你需要定义灯光的位置，以及照射的位置（默认位置是原点，故而此处省略了这一步）。然后 Three.js 通过用灯光位

置减去照射位置来计算出定向光的方向。在我们的示例中，灯光从屏幕的 `(0, 0, 1)` 位置指向 `(0，0，0)`，即直射位于原点位置的立方体。

为了看到灯光的效果，我们需要做一些事情。我们用一个 Phong 材质代替了上一个示例中使用的基础材质。在 Three.js 中，一个物体如何被照亮，不但依赖于添加到场景中的灯光，同时依赖于它自身的材质类型。Phong 材质类型实现了一个简单但看起来非常逼真的着色模型，简称 Phong *着色法*，它的性能非常优秀。我们现在可以看清立方体的边缘了：朝向灯光的面看起来更明亮，而背向灯光的面则看起来比较暗，明暗交界的边缘清晰地展现出两个面的相交处。关于光照还有更多的东西，但那些是更为基础的内容，我们会在下一章更详细地探讨光照的概念。现在至少我们创建了一个所谓的“真实效果 3D 模型”，仅使用一页的 JavaScript 代码。

> Phong 着色法由美国犹他大学的裴祥风（Bui Tuong Phong）发明。Phong 算法在当时是非常先进的，而如今也是许多渲染应用的一个标准渲染方式，尤其是在实时渲染中，以支持高效的真实明暗计算。如果想了解更多关于 Phong 着色法的信息，请访问其维基百科词条（http://en.wikipedia.org/wiki/Phong_shading）。

3.4　小结

本章我们介绍了 Three.js，最流行的基于 WebGL、用于构建 3D Web 应用的开源工具包。我们欣赏了一些使用它构建的令人惊叹的工程，从实验性的互动电影，到电子商务的可视化。我们从 GitHub 上获取了整个工程的最新源代码，并简单地浏览了它。最后，我们通过编写一些简单的程序来展现这个库带来的价值：用原生 WebGL 编写的需要几百代码的一段程序，使用 Three.js 只需几十行就可以实现。此外，Three.js 让我们得以基于完善的 3D 图形概念和熟悉的面向对象风格来工作。

本章大致讲述了 Three.js 是如何加快我们的开发速度的，在接下来的几章中，我们将具体了解 Three.js 的强大功能。

第 4 章

Three.js中的图形和渲染

在本章中，我们将体验 Three.js 为绘制图形和渲染场景提供的大量特性。如果你对 3D 编程还不熟悉，很有可能本章中的某些主题你无法马上理解。但是如果你一个个完成了代码示例，最终就可以很好地使用 Three.js 的能力来创造优秀的应用。

Three.js 有着丰富的图形系统，这一方面是受到了很多之前的 3D 图形库的启发，另一方面来源于它的作者们的经验。Three.js 提供了人们对 3D 代码库期望的特性，以及用多边形网格构建的 2D 和 3D 几何图形，包含层级结构物体和变换的场景图像，材质、纹理和灯光，实时阴影，开放给用户定义的可编程着色器，以及一个可用于高级特殊效果的、支持多通道和延迟渲染技术的灵活渲染系统。

4.1 几何图形和网格

比起直接使用 WebGL API 来工作，使用 Three.js 的一大便利在于它大大节省了我们用于构建几何形状的时间。回想一下第 2 章用于构建一个简单的立方体及其纹理贴图的大段代码，当时我们使用了 WebGL 缓冲，在绘制过程中，我们还需要编写一些代码来将这些数据转移到 WebGL 的内存空间中，以便进行真实的绘制。Three.js 提供了一系列现成的几何形状对象，将开发者从这些繁琐的工作中解救出来。这些几何形状包括预置的几何形状（如立方体和圆柱）、路径绘制形状、2D 挤出几何体，以及用户可扩展的基类，以供我们自由添加自定义的几何形状。下面让我们来探讨它们。

4.1.1 预置的几何形状类型

Three.js 提供了许多内置的常见几何形状。包括简单的立体，例如立方体、球体和圆柱；一些包含更复杂变量的形状，包括基础形状以及基于路径的形状、圆环和纽结；渲染在

3D 空间中的 2D 形状，例如圆、矩形和圆环；甚至还包括从 2D 文字计算转化而来的 3D 挤出文字形状。Three.js 同时支持绘制 3D 点和线。多数这类形状，你都可以使用一行构造函数很容易地创建出来，其中某些可能需要调整一些略微复杂的参数，以及编写一点点代码。

如果希望切实体验 Three.js 预置的几何形状，请运行 Three.js 项目中的 examples/webgl_geometries.html，如图 4-1 所示。示例中的每个网格对象包含一个代表一类内置的几何形状类型的形状，以及一个展示该形状纹理坐标计算方式的纹理贴图。纹理贴图来自 PixelCG Tips and Tricks，这是一个优秀的计算机图形指引教程站点。整个场景用一个水平灯光照亮，以展示每个物体的着色方式。

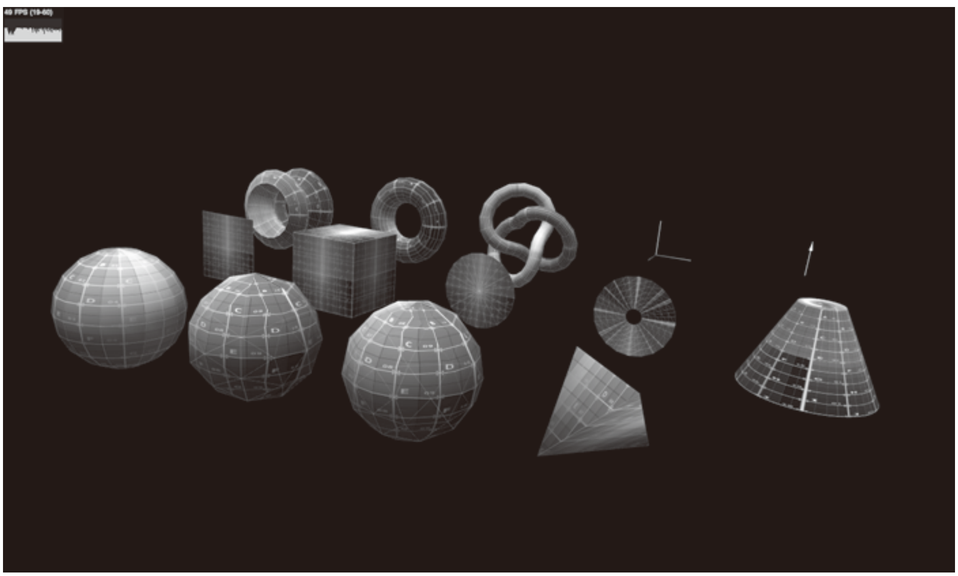


图 4-1：Three.js 内置几何形状演示。图中从左到右，从前到后依次为：球体、二十面体、八面体、四面体；平面、立方体、圆、环、圆台；车床、圆环、环面纽结、三维坐标系和方向矢量

4.1.2 路径、形状和挤出

Three.js 中的路径（Path）类、形状（Shape）类和挤出几何体（ExtrudeGeometry）类提供了许多灵活的几何图形生成方式，例如通过曲线来创建挤出几何体。图 4-2 展示了一个基于样条线的挤出形状。想要观察实际效果，请打开 Three.js 项目中的 examples/webgl_geometry_extrude_shapes.html。或者另一个例子，eamples/webgl_geometry_extrude_splines.html，这个示例允许你选择各种样条函数来生成挤出形状，并提供了用于查看曲线形状的运动相机。样条与挤出的结合是用于构建有机形状的重要技术，我们将在第 5 章对此进行详细说明。

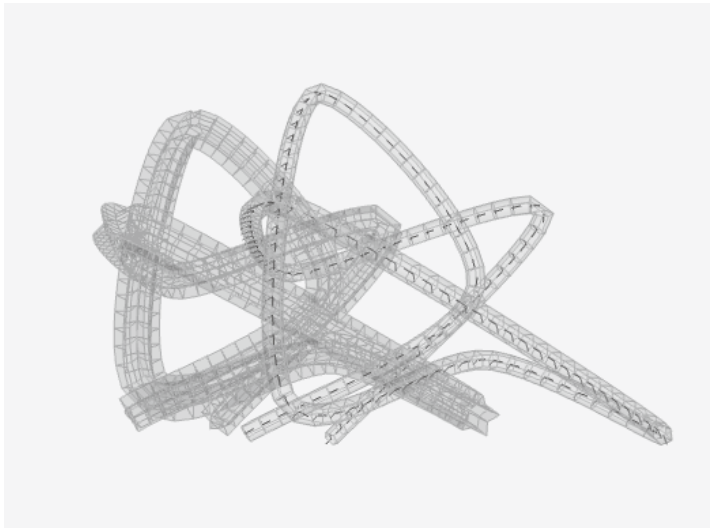


图 4-2：Three.js 中基于样条的挤出（另见彩插图 4-2）

`Shape` 类还可以用于创建 2D 平面形状或者基于这些形状的 3D 挤出。假设你有一个 2D 多边形数据的库（例如地理版图边缘或矢量艺术形状），你可以非常简单地利用 Three.js 提供的 `Path` 类将这些数据导入到 Three.js 环境中。`Path` 类同时提供了一些简单的路径绘制函数，例如 `moveTo()` 和 `lineTo()`，有 2D 绘图开发经验的人一定对这些函数非常熟悉（基本上，这是一套内置于 3D 绘图库中的 2D 绘图 API）。为什么要这样做呢？那是因为如果你有一个现成的 2D 形状，那么你可以利用它来创建一个存在于 3D 空间中的平面网格：它可以与其他 3D 物体同样的方式（平移、旋转、缩放）进行变换；它可以利用材质来进行绘制，并像场景中的其他物体一样被照亮和着色。你还可以对它进行挤出，从而创建一个基于 2D 外轮廓的真正的 3D 形状。

examples/webgl_geometry_shapes.html 中的演示，如图 4-3 所示，很好地展示了 Three.js 的这一能力。我们可以看到加利福尼亚州的轮廓、一些简单的多边形，以及特殊形状，如心形和笑脸。这些形状分别以多种方式绘制，包括平面 2D 网格、挤出并添加了切角的 3D 网格，以及线条——所有这些都是由基于路径的数据派生而来。

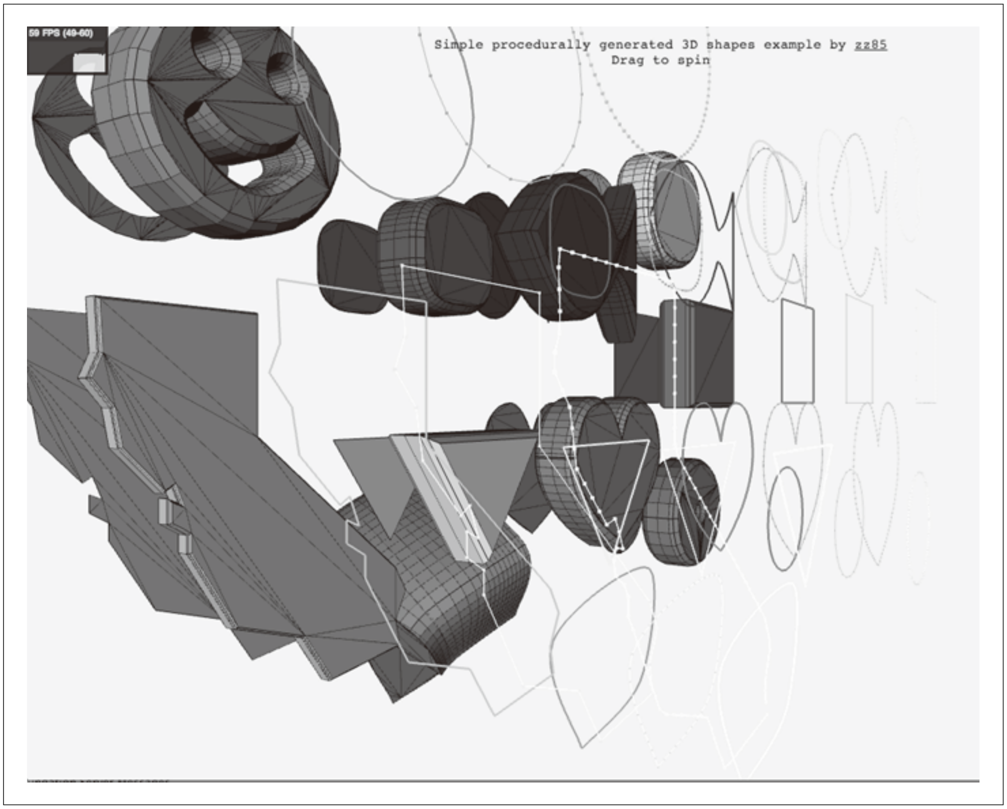


图 4-3：Three.js 实现基于路径的挤出形状（另见彩插图 4-3）

4.1.3 几何形状基础类

Three.js 预置的几何体类型派生自 THREE.Geometry（src/core/Geometry.js）基类。你也可以利用这个基类来编写你自己的几何形状。下面我们来看一看预置几何形状的源代码，以便对这些类如何实现几何形状有个大致的了解。几何形状类的代码存放在 Three.js 工程目录下的 src/extras/geometries/ 文件夹中。为了更好地说明，我们来浏览一下其中一个比较简单的几何物体，THREE.CircleGeometry 的实现代码。例 4-1 列出了这个物体的相关代码，总共耗费了一页的篇幅。

例 4-1：Three.js 圆形几何体代码

```
/**
 * @author hughes
 */

THREE.CircleGeometry = function ( radius, segments, thetaStart, thetaLength ) {

    THREE.Geometry.call( this );
    radius = radius || 50;
```

```
    thetaStart = thetaStart !== undefined ? thetaStart : 0;
    thetaLength = thetaLength !== undefined ? thetaLength : Math.PI * 2;
    segments = segments !== undefined ? Math.max( 3, segments ) : 8;

    var i, uvs = [],
    center = new THREE.Vector3(), centerUV = new THREE.Vector2( 0.5, 0.5 );

    this.vertices.push(center);
    uvs.push( centerUV );

    for ( i = 0; i <= segments; i ++ ) {

        var vertex = new THREE.Vector3();
        var segment = thetaStart + i / segments * thetaLength;

        vertex.x = radius * Math.cos( segment );
        vertex.y = radius * Math.sin( segment );

        this.vertices.push( vertex );
        uvs.push( new THREE.Vector2( ( vertex.x / radius + 1 ) / 2,
            ( vertex.y / radius + 1 ) / 2 ) );

    }

    var n = new THREE.Vector3( 0, 0, 1 );

    for ( i = 1; i <= segments; i ++ ) {

        var v1 = i;
        var v2 = i + 1 ;
        var v3 = 0;

        this.faces.push( new THREE.Face3( v1, v2, v3, [ n, n, n ] ) );
        this.faceVertexUvs[ 0 ].push( [ uvs[ i ], uvs[ i + 1 ], centerUV ] );

    }

    this.computeCentroids();
    this.computeFaceNormals();

    this.boundingSphere = new THREE.Sphere( new THREE.Vector3(), radius );

};

THREE.CircleGeometry.prototype = Object.create( THREE.Geometry.prototype );
```

`THREE.CircleGeometry` 的构造函数在 *xy* 平面上生成了一个平面圆形；这表示该形状中的所有的 *z* 值都被设成 0。整个算法的核心是用于计算这样一个形状的顶点数据的代码，这段代码包含在第一个 `for` 循环中：

```
        vertex.x = radius * Math.cos( segment );
        vertex.y = radius * Math.sin( segment );
```

事实上，这个 3D 圆形是由围绕中心旋转的多扇三角形构建而成的。三角形的数量越多，

构造出来的圆形就更接近于光滑的圆周。如图 4-4 所示。

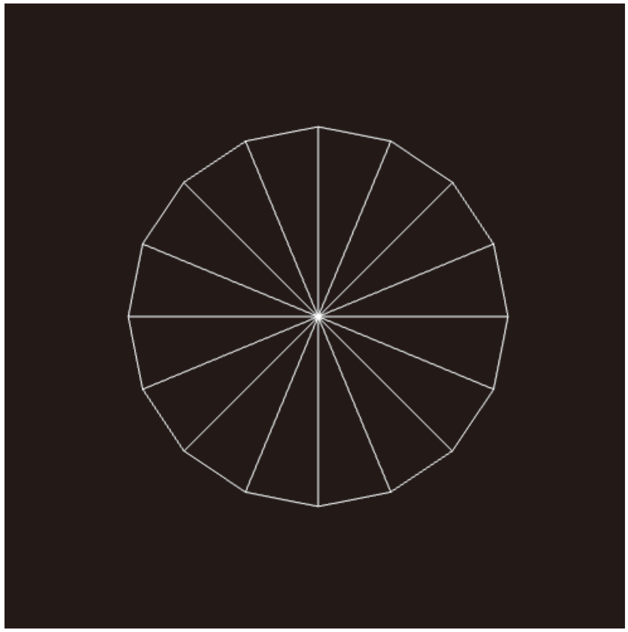


图 4-4：由三角形组成的 **THREE.CircleGeometry**

第一个循环体仅创建了圆周顶点位置的 *x* 和 *y* 坐标。现在我们需要为每个三角形创建一个面片（face，即多边形）对象，面片对象由三个顶点组成：定位于原点的中心点，以及圆周上的另外两个点。第二个循环进行了这一步工作，并将生成的数据填充到 `this.faces` 数组中。每个面片对象包含 `this.vertices` 数组中的三个顶点数据，分别以索引数 `v1`、`v2`、`v3` 来表示。注意，`v3` 的值始终为 0，这代表存储在 `this.vertices` 第一个元素的原点位置数据。（回忆第 2 章的 WebGL 细节，我们曾经使用 `gl.drawElements()` 方法来绘制三角形序列，三角形的顶点数据以索引数组的形式存储。在这里我们实现了同样的事情，不过这些细节被 Three.js 封装起来了。）

我们还应该注意到，每个循环中都包含了一个细节：纹理坐标的计算。WebGL 本身并不知道如何将纹理图片上的像素点映射到各个三角形上，除非我们告诉它应该如何做。与我们构建顶点值的方式类似，这两个 `for` 循环同时构建了纹理坐标，又称 UV 坐标（UV coordinates），并将这些数据存储在 `this.faceVertexUVs` 中。

回忆一下，纹理坐标是为每个顶点定义的一对浮点数，通常取值范围是 0 到 1。这些值对应位图数据上的 *x*、*y* 偏移量；着色器会利用这些数据从位图上获取像素信息。我们计算前两个顶点纹理坐标的方式和计算其顶点坐标数据的方式十分类似，都使用了旋转角度的余弦来计算 *x* 值，正弦来计算 *y* 值。不同之处在于纹理坐标通过除以圆的半径值来将值限定在 `[0..1]` 范围内。全部三角形第三个顶点的、对应几何图形原点位置的纹理坐标，在 2D 图像上的相应位置是图像的中心点 (0.5, 0.5)。

那么 UV 的含义是什么呢？字母 U 和 V 分别代表一张 2D 纹理图片的水平轴和垂直轴，因为 X、Y 和 Z 已经被用来表示物体 3D 坐标系统中的三轴。想要对 UV 坐标体系和 UV 映射（UV mapping）有更多的了解，请访问其维基百科词条（http://en.wikipedia.org/wiki/UV_mapping）。

在顶点和 UV 数据准备好之后，Three.js 便可以开始准备进行几何形状的渲染。`THREE.Circle` 构造函数结尾处的代码的作用是借助几何形状基类提供的辅助函数进行持久化操作。`computeCentroids()` 函数通过遍历几何形状的全部顶点并计算平均位置的方式来找到形状的几何中心点。

`computeFaceNormals()` 非常重要，因为物体的*法向量*（normal vector，简称 normal）决定了物体如何被着色。对一个平面的圆形来说，每个面片的法向量都垂直于几何形状本身。`computeFaceNormals()` 通过计算出来的垂直于组成圆的每个三角形所确定的平面的向量来定义圆的法向量。面片通常被描绘为一个平面阴影模式渲染的三角形，如图 4-5 所示。

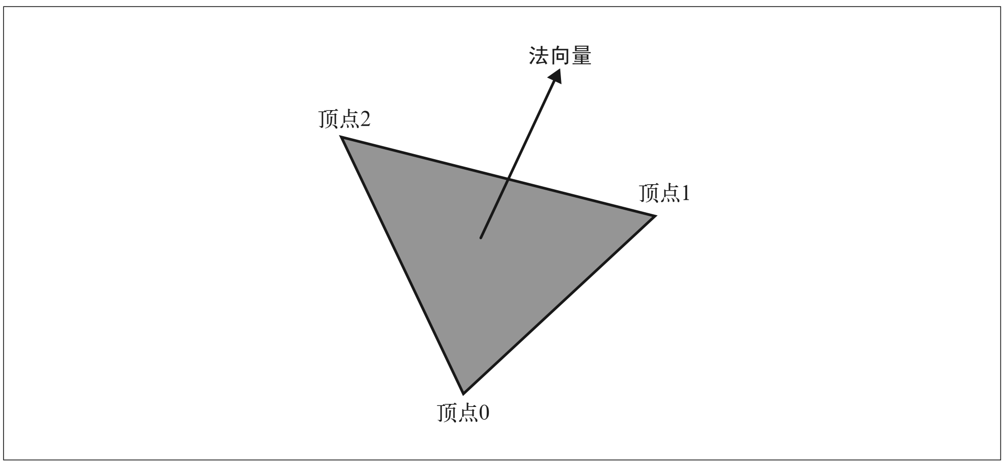


图 4-5：面片通常以一个平面阴影渲染的三角形来表示

最后，构造函数为物体初始化一个包围盒，在这个示例中，包围盒是一个球体，包围盒在拾取、选中以及图形渲染优化中起到重要作用。

4.1.4　用于优化网格渲染的BufferGeometry

Three.js 最近引进了一类名为 `THREE.BufferGeometry` 的优化版几何形状。`THREE.BufferGeometry` 将数据以类型数组的方式存储，这节省了用于处理 JavaScript 数字类型数组的额外开销。这个类也可以很方便地用于场景背景和支柱等顶点值不会发生改变并且不会在场景中移动的固定物体上。如果你确定一个物体是固定的，那么你可以为其创建一个 `THREE.BufferGeometry` 对象，Three.js 会进行一系列的优化来快速渲染这类物体。

4.1.5　从建模软件包中导入网格数据

到目前为止我们一直在研究如何用代码来创建几何形状。不过多数时候，我们的应用并不直接使用编程方式来构造几何形状，而是直接在程序中载入用专业的建模软件（例如 3ds Max、Maya 和 Blender）所构建的 3D 模型。

Three.js 提供了一系列用于转换和加载模型文件的静态函数。下面我们来观看一个加载网格的示例，包括其几何形状和材质。运行 Three.js 工程目录下的 examples/webgl_loader_obj_mtl.html 文件，你会看到图 4-6 所示的模型。

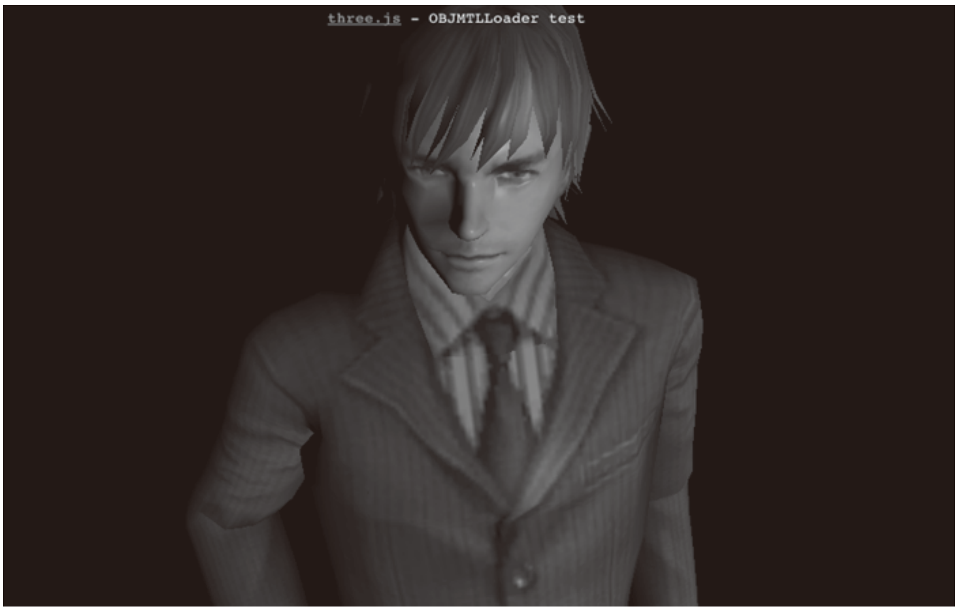


图 4-6：从 Wavefront OBJ 格式文件中加载的网格数据

这里描绘的男性形象通过 Wavefront OBJ 格式（文件后缀 .OBJ ）的文件导入。这是一个流行的、基于文本的模型格式，许多建模软件都支持导出这种格式的文件。OBJ 格式文件简单而有限，它仅仅包含几何形状的数据：顶点、法向量和纹理坐标。Wavefront 开发了另一种用于存储材质数据的配套文件格式 MTL，用于将材质和 OBJ 格式的模型数据进行关联。

Three.js 用于 OBJ 格式（包含材质）的加载器代码位于 examples/js/loaders/OBJMTLLoader.js。稍微研究一下它的工作机制，你就会发现，Three.js 的文件加载器使用预置的 geometry 类和 shape 类将载入的 OBJ 格式文件转换为 Three.js 中的 `THREE.Geometry` 几何形状对象，MTL 格式转换器将 MTL 格式的文件转换为 Three.js 可以识别的材质格式。生成的几何形状对象和材质对象最终被整合到一个 `THREE.Mesh` 对象中，并添加到场景。

Three.js 为多种文件格式提供了加载器示例。其中某些格式只提供了对几何形状和材质

的支持，但另一些可能更为复杂：包括整个场景、相机、灯光和动画。我们将在第 8 章对这些格式（以及用于编辑它们的工具）进行详细说明，第 8 章主要讲述创建内容的各种途径。

> Three.js 提供的多数文件加载器并不包含在核心的库代码中，而是包含在示例文件中。你需要在你的项目文件中另行引入加载器文件。除非特别注明，否则这些加载器和整个 Three.js 项目使用同样的开源许可证，你可以自由地使用它们。

4.2　场景图和空间变换的层级结构

WebGL 并没有原生的 3D 场景概念，它仅仅提供了将图像绘制到画布上的基础 API。场景的结构通常由应用自行提供。Three.js 基于成熟的*场景图*（scene graph）概念定义了一个结构化场景的模型。场景图代表一个以父子关系层级结构存储的 3D 物体集合，场景图的根节点通常用 root 变量来表示。应用从根节点开始递归地渲染场景的每个子层级。

4.2.1　利用场景图来管理复杂场景

场景图在表示具有层级结构的复杂物体时特别有用。想象一个机器人、一辆车，或者一个太阳系。这些物体都包含一系列的部件——四肢、轮子、卫星——这些部件都包含自有的行为。场景图同时允许这些物体按需被当作分离的组件或完整的组合来处理。这不仅仅是出于结构组织上的便利，它同时提供了一项被称为*变换层级*（transform hierarchy）的重要能力，在这个体系中，一个物体的子物体继承了父层级的变换信息（平移、旋转、缩放）。譬如，当你使用动画控制一辆小汽车的模型沿特定轨迹运动的时候，小汽车的整体沿轨迹运动，而小汽车的轮子同时在旋转。通过将轮子设置为小汽车主体的子元素，你可以在你的代码中动态地控制小汽车沿着轨迹运动，而轮子也将跟随小汽车主体在 3D 空间中沿同样轨迹运动；你无需另行为轮子编写沿轨迹运动的动画，只需设定它们的旋转动画。

> “图”这个词的使用，对于 Three.js 的场景图来说并不十分精确。在 3D 渲染中，场景图通常被表示为一个有向无环图（Directed Acyclic Graph，DAG）。在这种数据结构中，节点以父子关系存储，任何一个物体都可能同时拥有多个父级元素。在 Three.js 的场景图中，一个物体只能同时拥有一个父级元素。尽管出于技术术语统一的考虑，Three.js 的层级结构被称为一个“图”，但从更准确的角度来说，这是一个树结构。如果希望获得更多关于数学意义上的“图”的信息，请参考相应的维基百科词条（https://en.wikipedia.org/wiki/Directed_acyclic_graph）。

4.2.2　Three.js中的场景图

Three.js 中最基本的对象类型是 `THREE.Object3D`（Three.js 工程目录下的 src/core/Object3D.js 文件）。它是所有可见物体（如网格、线条、粒子系统）的基类，同时也是场景图层级结

构组织的基本要素。

每个 `Object3D` 对象都包含它自身的变换信息，这些信息分别存储在 `position`（位置，即平移）、`rotation` 和 `scale` 属性中。通过设置这些属性，你可以移动、旋转和缩放相应的物体。如果这个物体包含后代（子元素以及子元素的子元素），后代也将继承这些变换。如果后代同时具备自己的变换，那么这个变换将与父级元素的变换叠加生效，变换通过整个层级结构层层传递。我们来看一个例子。图 4-7 展示了一个非常简单的变换层级：`cube` 是 `cubeGroup` 的直接后代；`sphereGroup` 也是 `cubeGroup` 的直接后代（也就是说，它是 `cube` 的一个兄弟元素）；而 `sphere` 和 `cone` 都是 `sphereGroup` 的后代。

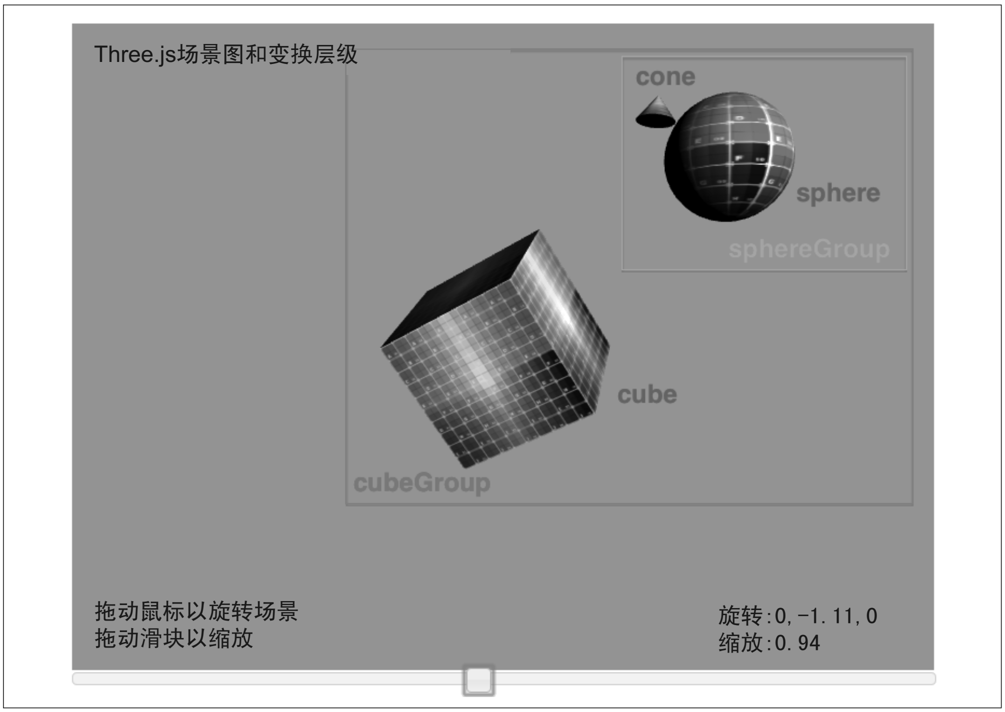


图 4-7：Three.js 场景图和变换层级

通过加载示例文件 Chapter 4/threejsscene.html 来运行这个示例。你会看到图中的立方体（cube）、球体（sphere）和圆锥（cone）分别在旋转。你可以用鼠标点击内容区域来旋转整个场景，或者拖动场景下方的滑块来对整个场景进行缩放。

例 4-2 展示了使用变换层级结构来创建和操作场景图的相关代码。重要的代码行用粗体标明。首先，我们通过创建一个 `Object3D` 对象 `cubeGroup` 来初始化整个场景。这个对象将作为整个场景图的根。随后我们将立方体（cube）网格以及另一个 `Object3D` 对象 `sphereGroup` 添加到 `cubeGroup` 对象中。球体（sphere）和圆锥（cone）被添加到 `sphereGroup` 对象中。同时我们将圆锥（cone）稍微往上挪了一些——通过设置它的 `position` 属性。

然后我们需要设置这些物体的动画：在 `animate()` 函数中，我们可以看到，当 `sphereGroup` 对象旋转的时候，它内部的球体也同时在进行自转。圆锥自身旋转的同时在空间中围绕球体公转。注意在每个动画帧的变化中，我们并没有为球体的自转和圆锥的公转单独编写代码，这两个物体都自动继承了父级元素 `sphereGroup` 对象的变换数据。类似地，实现整个整个场景的旋转和缩放交互也非常简单：只需设置 `cubeGroup` 对象的 `rotation` 和 `scale` 属性，这些变换在 Three.js 中就会自动冒泡，作用于它的后代元素。

例 4-2：一个包含变换层级的场景

```
function animate() {

        var now = Date.now();
        var deltat = now - currentTime;
        currentTime = now;
        var fract = deltat / duration;
        var angle = Math.PI * 2 * fract;

        // 绕y轴旋转立方体
        cube.rotation.y += angle;

        // 围绕其坐标y轴旋转球体分组
        sphereGroup.rotation.y -= angle / 2;

        // 围绕其坐标x轴旋转圆锥(向前翻滚)
        cone.rotation.x += angle;
    }

        function createScene(canvas) {

        // 创建Three.js渲染器并将其添加到canvas中
        renderer = new THREE.WebGLRenderer( { canvas: canvas, antialias: true } );

        // 设置视口尺寸
        renderer.setSize(canvas.width, canvas.height);

        // 创建一个Three.js场景
        scene = new THREE.Scene();

        // 添加一个相机以便观察整个场景
        camera = new THREE.PerspectiveCamera( 45, canvas.width / canvas.height,
            1, 4000 );
        camera.position.z = 10;
        scene.add(camera);

        // 创建一个用于容纳所有物体的分组
        cubeGroup = new THREE.Object3D;

        // 添加一个相机以便观察整个场景
        var light = new THREE.DirectionalLight( 0xffffff, 1.5);
        // 将灯光放置在场景外,指向原点
        light.position.set(.5, .2, 1);
        cubeGroup.add(light);

        // 为立方体创建一个带纹理的Phong材质
```

```
    // 首先,创建纹理贴图
    var mapUrl = "../images/ash_uvgrid01.jpg";
    var map = THREE.ImageUtils.loadTexture(mapUrl);
    var material = new THREE.MeshPhongMaterial({ map: map });

    // 创建立方体几何形状
    var geometry = new THREE.CubeGeometry(2, 2, 2);

    // 其次,将材质和几何形状整合到一个网格中
    cube = new THREE.Mesh(geometry, material);

    // 将网格向观察者倾斜
    cube.rotation.x = Math.PI / 5;
    cube.rotation.y = Math.PI / 5;

    // 将立方体网格添加到分组中
    cubeGroup.add( cube );

    // 为球体创建一个分组
    sphereGroup = new THREE.Object3D;
    cubeGroup.add(sphereGroup);

    // 将球体分组移动到立方体的后上方
    sphereGroup.position.set(0, 3, -4);

    // 创建球体几何形状
    geometry = new THREE.SphereGeometry(1, 20, 20);

    // 将材质和几何形状整合到一个网格中
    sphere = new THREE.Mesh(geometry, material);

    // 将球体网格添加到分组中
    sphereGroup.add( sphere );

    // 创建圆锥几何形状
    geometry = new THREE.CylinderGeometry(0, .333, .444, 20, 5);

    // 接着将材质和几何形状整合到一个网格中
    cone = new THREE.Mesh(geometry, material);

    // 将圆锥移动到球体的上方,并和球体保持一定距离
    cone.position.set(1, 1, -.667);

    // 将圆锥网格添加到分组中
    sphereGroup.add( cone );

    // 现在将分组添加到场景中
    scene.add( cubeGroup );
}

function rotateScene(deltax)
{
    cubeGroup.rotation.y += deltax / 100;
    $("#rotation").html("rotation: 0," + cubeGroup.rotation.y.toFixed(2) + ",0");
}
```

```
function scaleScene(scale)
{
    cubeGroup.scale.set(scale, scale, scale);
    $("#scale").html("scale: " + scale);
}
```

4.2.3 平移、旋转和缩放的表示

在 Three.js 中，变换是通过 3D 矩阵计算来实现的，所以很自然，`Object3D` 的变换要素使用三维向量表示：位置（`position`）、旋转（`rotation`）和缩放（`scale`）。`position` 的含义非常明显：*x*、*y* 和 *z* 要素分别定义了物体到原点的位置偏移。`scale` 的含义也非常直观：*x*、*y* 和 *z* 元素分别用于表示物体在三个维度上的缩放程度。

`rotation` 的几个要素的含义需要稍微解释一下：*x*、*y* 和 *z* 分别表示物体围绕相应坐标轴的旋转弧度；举例来说，`(0, Math.PI / 2, 0)` 表示物体围绕自身的 *y* 轴旋转了 90 度。（注意旋转角度用弧度来表示，2π 弧度等于 360 度）。这种旋转信息的表示方式——使用围绕 *x*、*y*、*z* 三轴的旋转角度来组合表示整个物体旋转的方式——被称为*欧拉角*（Euler angle）。我想，Mr.doob 之所以采用欧拉角来作为默认的表示方式，是由于这种方式非常直观并且便于处理；尽管如此，在实践中欧拉角这种表示方式还是会遇到很多计算处理上的问题，所以 Three.js 同时支持使用四元数来表示旋转。这是区别于欧拉角的另一种旋转表示形式，相对而言，它需要编写更多代码来处理旋转运算相关的事务。四元数非常精确，但操作起来并不直观。

在 Three.js 内部，每个 `Object3D` 都持有一个用于存放各种变换信息的矩阵。拥有多重祖先元素的物体递归地用祖先元素的变换矩阵来乘自身的变换矩阵，从而得到自身最终的变换结果；也就是说，Three.js 在每次场景渲染的时候都遍历整个场景图（树结构）的叶节点来计算每个物体的变换矩阵。Three.js 为 `Object3D` 对象定义了一个 `matrixAutoUpdate` 属性，当这个属性被设为 `false` 的时候，上述的行为（自动继承父级元素变换）被禁止。尽管如此，这个特性很有可能会引发一些奇怪的 bug（“为什么我的动画不刷新了？”），所以请谨慎使用它。

4.3 材质

我们在 WebGL 应用中看到的可见形状都持有一些用来表示外观的属性，如颜色、着色方法以及纹理（位图）。如果使用底层的 WebGL API 来构建这些属性，我们需要自行编写 GLSL 着色器代码，这需要相对高级的编程技巧，就算你需要创建的仅仅是最简单的视觉效果。所幸 Three.js 预先为我们准备好了 GLSL 代码，并封装在*材质*（material）对象中。

4.3.1 标准网格材质

回忆一下，WebGL 需要开发者提供一个可编程着色器，用于每个物体的绘制。你可能已经注意到了，在本章中，至今为止我们并没有提及任何与 GLSL 着色器源代码相关的内容。原因在于，Three.js 已经为我们预先编写好了“开箱即用”的着色器代码，这些着色

器代码以一个预置的 GLSL 代码库的形式提供。

典型的场景图库和流行的建模工具通常将着色器作为材质的一部分来实现，材质是一个定义了 3D 网格、点或者线元外观属性的对象，这些外观属性包括颜色（color）、透明度（transparency）以及发光（shininess）。材质还包含（也可以不包含）纹理贴图，这是一个覆盖物体表面的位图。材质属性结合网格的顶点数据、场景的光照信息，可能还包括相机位置信息和其他全局属性，来决定每个物体的最终渲染效果。

Three.js 提供了预置类 `MeshBasicMaterial`、`MeshPhongMaterial` 和 `MeshLambertMaterial` 来支持常用的材质类型。（`Mesh` 前缀表示这些材质类型可以用于网格对象。相对地，其他类型的对象，如线条对象和粒子对象，也有其相应的适用材质类型。查看 Three.js 下的 src/materials 目录，以了解最新的材质集合。）这些材质类型使用三种著名的材质技术来实现。

- *unlit（又称 prelit）*

  当使用这种材质类型时，仅纹理、颜色和透明度属性会被用于物体的外观渲染。场景的光照对物体的外观不会产生影响。这是用于扁平风格渲染效果或者绘制无需复杂着色效果的简单几何形状的绝佳材质类型。这种材质对已经将光照信息和材质结合计算并预先输出到纹理中（例如使用 3D 建模工具创建并使用了烘培贴图）的情况同样适用，因为此时物体的外观效果无需由渲染器再次进行计算。

- *Phong 着色法*

  这种材质提供了一个简单而优秀的仿真风格的高性能着色模型。它已经成为迅速而简便地实现明暗视觉效果的常规方法，同时许多游戏和应用也还在使用这种着色方法。用 Phong 着色法渲染的物体会在光线直接照射的地方显示高光区域（镜面反射），物体表面的亮度随各点到光源的距离产生衰减，而背光区域会被渲染成完全黑暗的效果。

- *朗伯反射（Lambertian reflectance）*

  在 Lambert 着色法中，物体外观的明暗不随观察者视角的改变而改变。它非常适合于用来表现云朵等会漫反射大部分光线的物体，或者像月球这类具有高反射率（表面反射光非常明亮）的卫星。

运行本书示例中的 Chapter 4/threejsmaterials.html，你可以更直观地感受 Three.js 材质的不同类型。在如图 4-8 所示的页面中，展示了一个带月球表面纹理贴图的、被照亮的球体。月球是用于描绘各材质类型差异的绝佳素材。例如点击单选按钮，在 Phong 和 Lambert 材质间切换，对比物体使用 Lambert 着色方法相比使用 Phong 着色方法看起来是否更合理了。此外使用 Basic（unlit）着色器，观察仅仅使用纹理渲染而不考虑光照的情况下球体的外观。

尝试改变漫反射和镜面反射的颜色，观察其效果。材质的漫反射颜色定义了物体在一个方向上能反射多少来自光源的光线，包括定向光、点光源和聚光灯（我们会在下一节中详细讨论这些灯光类型）。镜面反射颜色与场景灯光合成创建出物体表面正对光源的顶点高光效果。（注意镜面高光仅仅在使用了 Phong 材质的前提下生效，其他类型的材质不支持镜面反射颜色。）此外，尝试关闭纹理贴图，你可以更直观地观察这种材质在简单球体上产生的效果。最后，勾选“wireframe”选项来观察这些设置对线框渲染产生了何种影响。

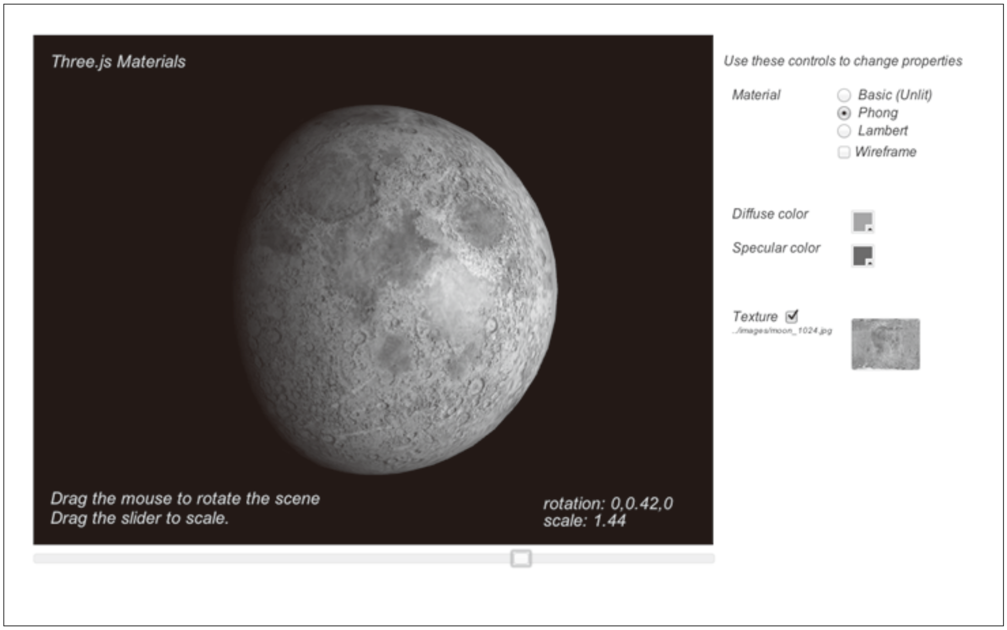


图 4-8：Three.js 标准网格材质类型——Basic (unlit)、Phong 和 Lambert（另见彩插图 4-8）

4.3.2 使用多重纹理增添逼真效果

上面的示例展示了材质是如何用于定义一个物体的外观的。Three.js 的大多数材质类型都支持使用多重纹理来为物体创建更逼真的效果。允许在一个材质中使用多个纹理，或者说多重纹理的意图在于，提供一个低成本的仿真计算方案——配合使用更多多边形来表示物体，或使用多道渲染工序来反复处理物体的方案。这里提供了一些例子，用于展示 Three.js 支持的常见多重纹理贴图技术。

凹凸贴图。凹凸贴图（bump map）用于替代物体表面的法向量来为表面提供凹凸的视觉效果。位图的像素信息被当作高度而非色彩信息来进行处理。举例来说，一个值为 0 的像素点代表没有进行任何偏移，而一个非零的值可以用来表示相对物体表面的正向偏移。通常，单通道的黑白位图可用于节省性能开销，而完整的、可以存储更多数值信息的 RGB 通道位图则可以用来表示更多的细节。使用位图来代替 3D 向量的原因是位图更紧凑，并且为着色器内部代码提供了一个用于计算正位移的更高效的方法。打开示例文件 example Chapter 4/threejsbumpmap.html，实际感受一下凹凸贴图的效果，如图 4-9 所示。打开 / 关闭月球的纹理，并改变漫反射和镜面反射颜色来观察不同的效果，你也许已经注意到，尽管效果非常酷，它也可能产生不佳的效果 。但不管怎么说，凹凸贴图为添加真实细节提供了一个低成本的方法。

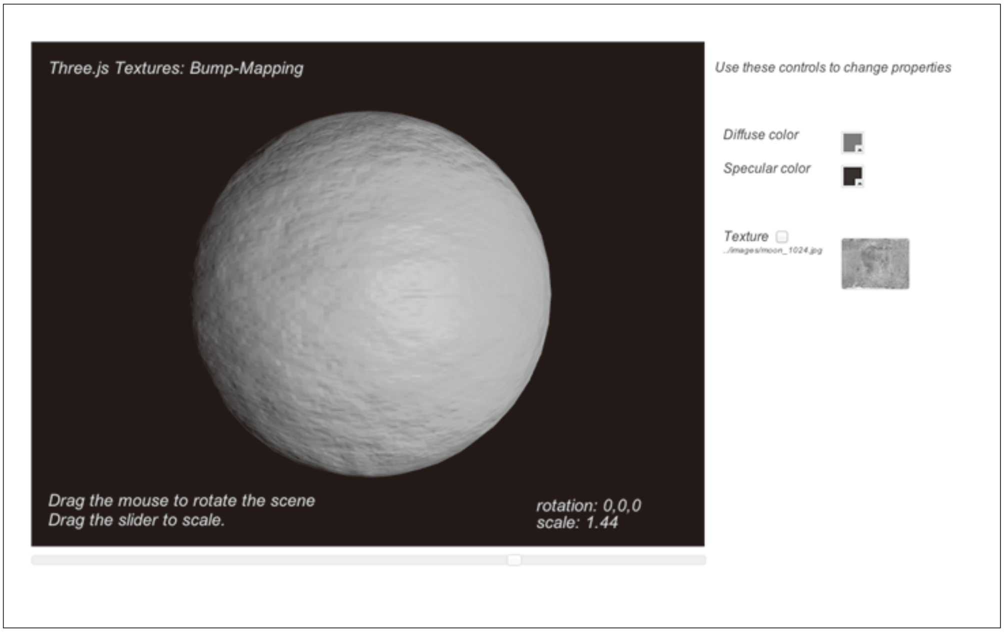


图 4-9：凹凸贴图（另见彩插图 4-9）

在 Three.js 中使用凹凸贴图非常简单。只需在 THREE.MeshPhongMaterial 构造器初始化的时候将传入参数的 bumpMap 属性设置为一个有效纹理即可：

```
material= new THREE.MeshPhongMaterial({map: map,
  bumpMap: bumpMap });
```

法向量贴图。法向量贴图（normal map）提供了一个能够比凹凸贴图展现更多细节的方式，无需使用更多的多边形。相比凹凸贴图，法向量贴图的体积更大，对计算性能的要求也更高。法向量贴图的工作原理是将实际的顶点法向量信息编码到 RGB 格式位图的数据中，这通常会比相应的网格顶点数据有更高的分辨率。着色器将法向量信息和光照计算（包括综合处理当前的相机和光源信息）相结合最终渲染出物体的外观细节。打开示例文件 Chapter 4/threejsnormalmap.html，查看法向量贴图的实际效果。法向量位于图 4-10 的右下角。注意地球模型的海拔信息轮廓。打开 / 关闭法向量贴图来对比法向量贴图是如何为地球模型提供细节的；你会惊叹于区区一个位图竟能为如球体这样简单的模型提供如此之多的细节信息。

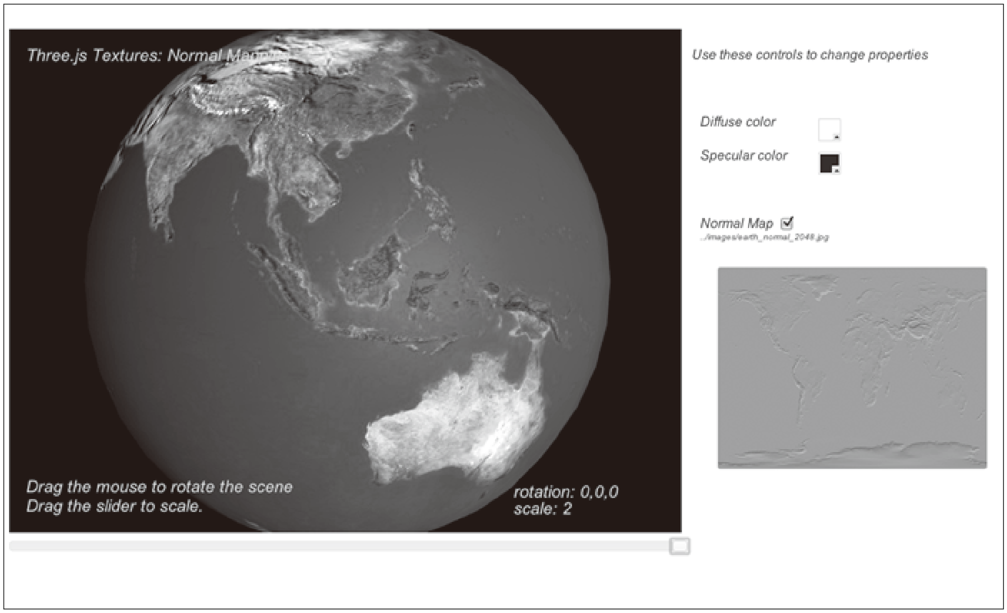


图 4-10：使用法向量贴图的地球

在 Three.js 中使用法向量贴图同样十分简单。只需在构造 `THREE.MeshPhongMaterial` 对象的时候，将一个已有的纹理作为 `normalMap` 属性配置传入即可：

```
Material = new THREE.MeshPhongMaterial({ map: map,
  normalMap: normalMap });
```

环境贴图。环境贴图（environment map）为使用其他纹理来提升真实度提供了另一种途径。环境贴图模拟了物体对周围环境的反射，而不是像凹凸贴图和纹理贴图那样旨在为物体表面添加更多的真实细节。

打开 Chapter 4/threejsenvmap.html 查看环境贴图的示例。在内容区域拖拽鼠标来旋转场景，或者使用鼠标滚轮对场景进行放大缩小。注意球表面的图像呈现为它周围天空背景的贴图（如图 4-11）。事实上，它并没有真正进行这样的处理，整个场景的背景由一个立方体的内侧表面构成，而球体只是简单地将用于场景背景的纹理像素渲染到自身的表面上。诀窍在于，用于球体表面的材质是一个*立方体纹理*（cube texture）材质：这是一个由六个不同的位图组成的、可以在立方体内部形成一个连续图像效果的纹理贴图。这个特殊的立方体材质被用于构建一个天空全景的背景。查看 images/cubemap/skybox/ 文件中的不同文件，了解这个场景是如何被构建出来的。由于采用了立方体纹理，这种类型的环境贴图被称为*立方体环境映射*（cubic environment mapping）。

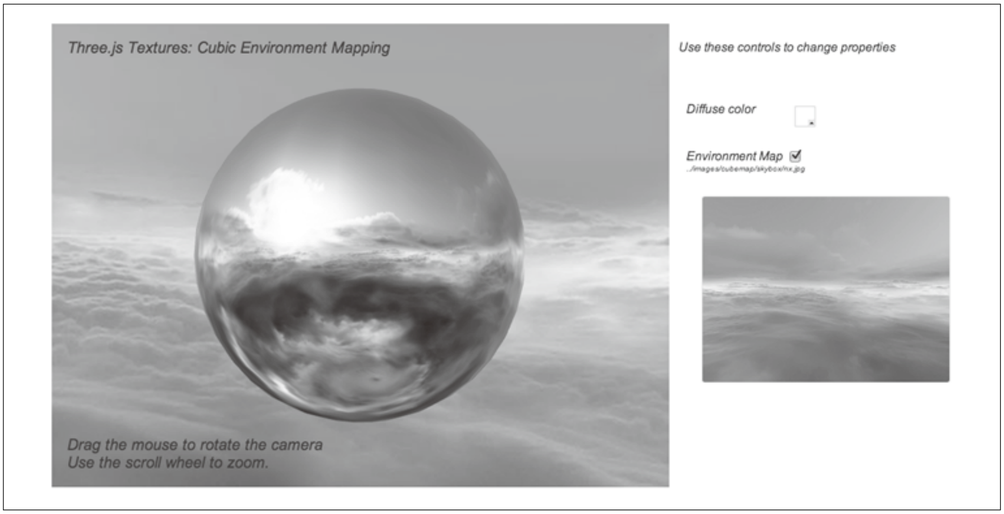


图 4-11：立方体环境贴图，具有逼真场景背景和反射效果

在 Three.js 中使用立方体纹理比使用凹凸贴图或法向量贴图要稍微复杂一些，首先，我们需要创建一个立方体纹理，而非普通的纹理。我们需要使用 Three.js 的全局方法 `ImageUtils.loadTextureCube()`，通过传入六个不同图像的 URL 来创建这个纹理。然后，我们在创建 `MeshPhongMaterial` 对象的时候将 `envMap` 参数设置为该纹理对象。我们还指定了反射率（`reflectivity`）的值，用来定义立方体纹理在物体渲染时被物体表面材质“反射”的程度。在这个例子中，我们指定了稍微大于默认值 1 的值，以便让环境贴图看起来更明显些。

```
var path = "../images/cubemap/skybox/";

var urls = [ path + "px.jpg", path + "nx.jpg",
             path + "py.jpg", path + "ny.jpg",
             path + "pz.jpg", path + "nz.jpg" ];

envMap = THREE.ImageUtils.loadTextureCube( urls );
materials["phong-envmapped"] = new THREE.MeshBasicMaterial(
   { color: 0xffffff,
     envMap : envMap,
     reflectivity:1.3} );
```

为了使它更贴近于真实的效果，我们还需要做一些事情。我们需要将反射位图和周围环境进行绑定。为此我们创建了一个天空盒（skybox），这是一个内面贴图的背景立方体，使用了同样的位图图像，用于构建一个天空全景。这本身需要非常繁杂的工作，不过所幸 Three.js 提供了内建的辅助函数来帮助我们完成这件事情。除了内置的标准材质 Basic、Phong 和 Lambert 以外，Three.js 还提供了一个着色器的库，包含在全局变量 `THREE.ShaderLib` 中。我们使用立方体（cube geometry）来创建一个网格，并使用库中预先定义的“cube”着色器来作为立方体的材质。它使用我们刚才用于环境贴图的纹理，来渲染立方体的内部。

```
// 创建天空盒
var shader = THREE.ShaderLib[ "cube" ];
shader.uniforms[ "tCube" ].value = envMap;

var material = new THREE.ShaderMaterial( {

    fragmentShader: shader.fragmentShader,
    vertexShader: shader.vertexShader,
    uniforms: shader.uniforms,
    side: THREE.BackSide

} ),

mesh = new THREE.Mesh(new THREE.CubeGeometry( 500, 500, 500 ), material);
scene.add( mesh );
```

4.4 光源

光源照亮 3D 场景中的物体。Three.js 定义了多种内置的光源类型，这些光源类型对应建模工具和其他场景图库中定义的光源类型。最常用的光源类型有*定向光*（directional light）、*点光源*（point light）、*聚光灯*（spotlight）和*环境光*（ambient light）。

- *定向光*

  实现了一类产生定向平行光的光源。这类光源没有位置信息，只有方向、颜色和强度信息。（事实上，在 Three.js 里面，定向光是包含一个位置信息的，不过它仅仅被用来结合另一个向量——目标位置——来计算光线的方向。这显然是一个并不高明并违反直觉的处理方式，我希望 Mr.doob 后续能够修复它。）

- *点光源*

  包含位置信息，但不包含方向信息。

- *聚光灯*

  同时具备位置和方向信息。同时它们提供了用于定义聚光灯内圆锥和外圆锥尺寸（角度）的参数，以及定义它们能够照亮多远距离的参数。

- *环境光*

  没有位置，也没有方向。它均匀地投射于整个场景。

所有的 Three.js 光源类型都支持通用的 `intensity` 属性，它定义了光源的强度、颜色和 RGB 值。

光源自身并没有处理对场景的影响，它们的值和特定材质的属性结合起来，用来定义物体最终的视觉展示效果。`MeshPhongMaterial` 和 `MeshLambertMaterial` 定义了以下属性。

- `color`

  又称漫反射颜色（diffuse color），它定义了物体是如何反射来自一个方向上的光的（例如定向光、点光源和聚光灯）。

- `ambient`

  物体对环境光的反射程度。

- `emissive`

  这个材质属性定义了一个物体自身发光的颜色，和整个场景的光源无关。

`MeshPhongMaterial` 材质同时支持一个镜面反射（`specular`）颜色属性，这个属性和场景灯光相结合，创建出物体朝向光源的顶点的高光效果。

回忆一下，`MeshBasicMaterial` 材质会忽略所有灯光的影响。

图 4-12 展示了一个使用 Three.js 灯光类型创建的灯光实验。打开 Chapter 4/threejslights.html 运行这个示例。这个场景包含四个不同类型的灯光、黑白纹理的背景平面和三个纯白的几何体，用来展示不同灯光的效果。你可以通过页面上的拾色器控件动态地改变每个灯光的颜色。如果你把一个灯光的颜色设为黑色，那么这个灯光将会被完全关闭。在内容区域拖拽鼠标来旋转整个场景，并观察灯光对模型不同部分造成的影响。

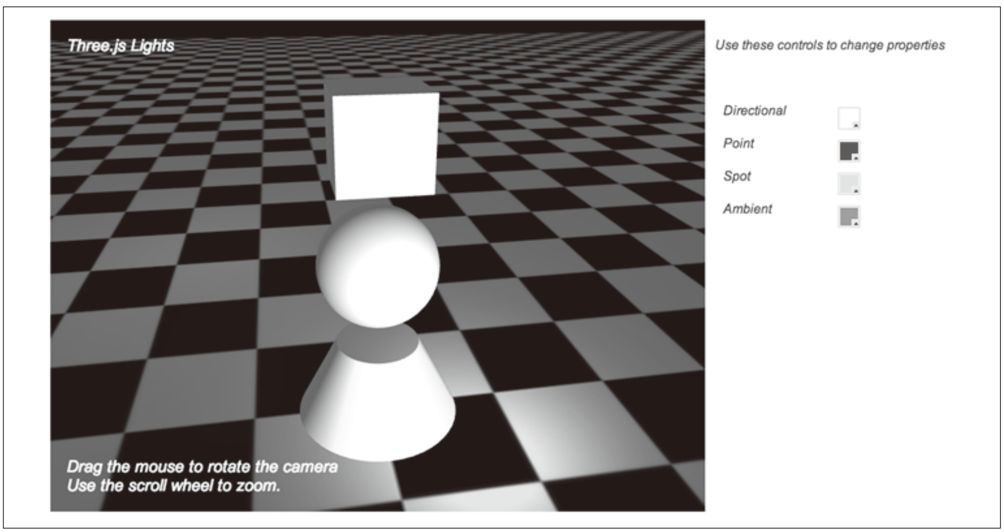


图 4-12：定向光、点光源、聚光灯和环境光（另见彩插图 4-12）

下面这段代码展示了灯光的设置。位于场景前方的白色定向光照亮了几何体前方的白色区域。蓝色的点光源从后方照亮物体模型；注意物体后方地板上的蓝色区域。通过定义 `spotLight.target.position` 属性，绿色的聚光灯的锥体笼罩了靠近场景前方的地板区域。最后，环境光平均地为场景中的所有物体提供了少量的亮度。使用控件来更改设置，并从各个角度观察场景，以更直观地了解各类灯光各自以及叠加起来产生的效果。

```
// 创建并将所有灯光添加到场景中
directionalLight.position.set(.5, 0, 3);
root.add(directionalLight);

pointLight = new THREE.PointLight (0x0000ff, 1, 20);
pointLight.position.set(-5, 2, -10);
```

```
root.add(pointLight);

spotLight = new THREE.SpotLight (0x00ff00);
spotLight.position.set(2, 2, 5);
spotLight.target.position.set(2, 0, 4);
root.add(spotLight);

ambientLight = new THREE.AmbientLight ( 0x888888 );
root.add(ambientLight);
```

这里做个友情提示，和 WebGL 中的其他东西一样，灯光是一个完全人为创造的概念。WebGL 只能处理缓冲和着色器；开发者需要通过编写着色器代码来合成灯光效果。Three.js 提供了令人震惊的强大的材质和灯光能力……尤其是当你意识到这些完全是通过 JavaScript 代码来编写的时候。当然，如果 WebGL 没有提供访问 GPU 的能力，这些都将无法实现。

4.5 阴影

一直以来，设计师都使用阴影来营造更加真实的视觉效果。通常这些阴影都是伪造的、预渲染的效果，移动场景中的灯光或者移动带阴影的物体，都会破坏这种效果。然而，Three.js 支持根据当前灯光和物体的位置进行实时阴影渲染。

Chapter 4/threejsshadows.html 中的示例演示了如何在场景中增加实时阴影。如图 4-13：位于地板上方、场景前方的聚光灯使物体在地面上产生了阴影。注意阴影是如何随着立方体的旋转变化的。同时，当地板旋转时，阴影并没有跟随地板运动。如果阴影是用预渲染伪造的，那么它会“粘”在地板上，并且不会随着立方体的旋转而变化。通过控件来更改灯光设置，尤其是聚光灯，来观察阴影是如何动态变化的。

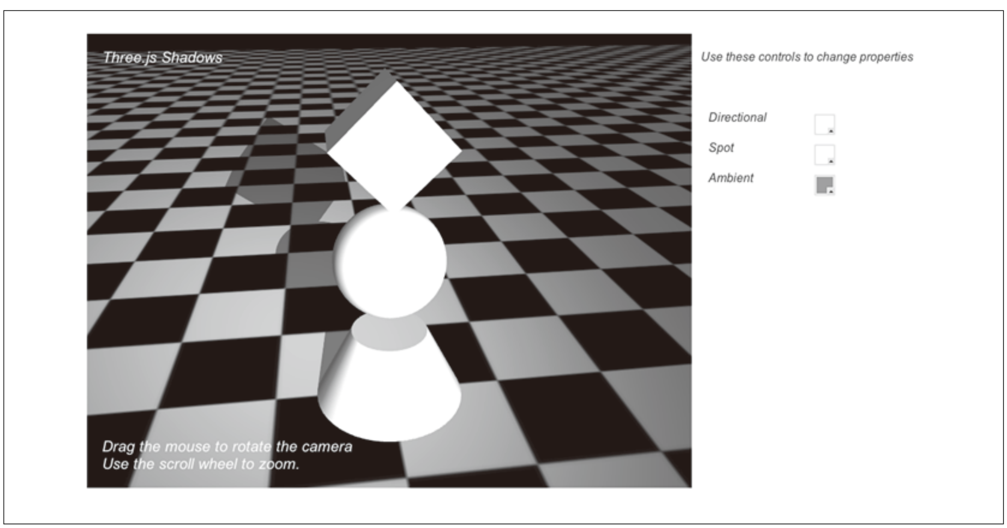


图 4-13：使用聚光灯和阴影贴图营造实时阴影

Three.js 使用一项名为阴影贴图（shadow mapping）的技术来支持阴影效果。渲染器会为阴影贴图保存一张额外的纹理，渲染器会将阴影区域渲染到这个纹理中，并在片段着色器中将其用于合成最终的图像。故而，在 Three.js 中开启阴影需要以下步骤。

(1) 在渲染器中开启阴影贴图。
(2) 在灯光中开启阴影并设置相关参数。THREE.DirectionalLight 和 THREE.SpotLight 两种类型都支持阴影。
(3) 设置哪些物体会产生和接受阴影。

让我们来看看这些步骤在代码中是如何实现的。例 4-3 中用粗体高亮的代码展示了在 createScene() 函数中添加的用于渲染阴影的代码。

例 4-3：Three.js 中的阴影贴图

```
var SHADOW_MAP_WIDTH = 2048, SHADOW_MAP_HEIGHT = 2048;

function createScene(canvas) {

    // 创建Three.js渲染器并将其添加到canvas中
    renderer = new THREE.WebGLRenderer( { canvas: canvas, antialias: true } );

    // 设置视口尺寸
    renderer.setSize(canvas.width, canvas.height);

    // 开启阴影
    renderer.shadowMapEnabled = true;
    renderer.shadowMapType = THREE.PCFSoftShadowMap;

    // 创建Three.js场景
    scene = new THREE.Scene();

    // 添加一个相机以便观察整个场景
    camera = new THREE.PerspectiveCamera( 45, canvas.width / canvas.height,
        1, 4000 );
    camera.position.set(-2, 6, 12);
    scene.add(camera);

    // 创建一个用于容纳所有物体的分组
    root = new THREE.Object3D;

    // 添加一个相机以便观察整个场景
    directionalLight = new THREE.DirectionalLight( 0xffffff, 1);

    // 创建并将所有灯光添加到场景中
    directionalLight.position.set(.5, 0, 3);
    root.add(directionalLight);

    spotLight = new THREE.SpotLight (0xffffff);
    spotLight.position.set(2, 8, 15);
    spotLight.target.position.set(-2, 0, -2);
    root.add(spotLight);

    spotLight.castShadow = true;
```

```
spotLight.shadowCameraNear = 1;
spotLight.shadowCameraFar = 200;
spotLight.shadowCameraFov = 45;

spotLight.shadowDarkness = 0.5;

spotLight.shadowMapWidth = SHADOW_MAP_WIDTH;
spotLight.shadowMapHeight = SHADOW_MAP_HEIGHT;

ambientLight = new THREE.AmbientLight ( 0x888888 );
root.add(ambientLight);

// 创建一个用于容纳球体的组
group = new THREE.Object3D;
root.add(group);

// 创建一个纹理贴图
var map = THREE.ImageUtils.loadTexture(mapUrl);
map.wrapS = map.wrapT = THREE.RepeatWrapping;
map.repeat.set(8, 8);

var color = 0xffffff;
var ambient = 0x888888;
// 添加一个作为平面的地面,以便更好地观察灯光
geometry = new THREE.PlaneGeometry(200, 200, 50, 50);
var mesh = new THREE.Mesh(geometry, new THREE.MeshPhongMaterial({color:color,
    ambient:ambient, map:map, side:THREE.DoubleSide}));
mesh.rotation.x = -Math.PI / 2;
mesh.position.y = -4.02;

// 将网格添加到分组中
group.add( mesh );
mesh.castShadow = false;
mesh.receiveShadow = true;

// 创建立方体几何形状
geometry = new THREE.CubeGeometry(2, 2, 2);

// 然后将几何形状和材质整合到网格中
mesh = new THREE.Mesh(geometry, new THREE.MeshPhongMaterial({color:color,
    ambient:ambient}));
mesh.position.y = 3;
mesh.castShadow = true;
mesh.receiveShadow = false;

// 将网格添加到分组中
group.add( mesh );

// 将网格持久化到变量中以便对其进行旋转操作
cube = mesh;

// 创建球体几何形状
geometry = new THREE.SphereGeometry(Math.sqrt(2), 50, 50);

// 然后将几何形状和材质整合到网格中
```

```
        mesh = new THREE.Mesh(geometry, new THREE.MeshPhongMaterial({color:color,
            ambient:ambient}));
        mesh.position.y = 0;
        mesh.castShadow = true;
        mesh.receiveShadow = false;

        // 将网格添加到分组中
        group.add( mesh );

        // 创建圆锥几何形状
        geometry = new THREE.CylinderGeometry(1, 2, 2, 50, 10);

        // 然后将几何形状和材质整合到网格中
        mesh = new THREE.Mesh(geometry, new THREE.MeshPhongMaterial({color:color,
            ambient:ambient}));
        mesh.position.y = -3;

        mesh.castShadow = true;
        mesh.receiveShadow = false;

        // 将网格添加到分组中
        group.add( mesh );

        // 现在将分组添加到场景中
        scene.add( root );

    }
```

首先，我们通过设置 renderer.shadowMapEnabled 为 true 并将其 shadowMapType 属性设置为 THREE.PCFSoftShadowMap 来开启阴影。Three.js 支持三种类型的阴影贴图算法：基础（basic）、PCF（percentage close filtering）和 PCF soft shadows。每种算法都比前一种更接近真实效果，但会导致复杂度的上升和性能的下降。尝试在这个示例中将 shadowMapType 的值改为 THREE.BasicShadowMap 和 THREE.PCFShadowMap 来查看效果，阴影的质量在低质量的设定下会明显降低。但如果你的场景非常复杂，那么你就得通过降低质量的方式来提升性能。

接下来，我们需要为聚光灯开启阴影。我们将其 castShadow 属性设置为 true。同时我们需要设置 Three.js 所需的其他几个参数。Three.js 通过从灯光位置引一条通过目标物体的射线的方式来渲染阴影。从本质上说，它将聚光灯当成另一种类型的“相机”来处理。所以我们需要设置一些类似相机设置的参数，包括近和远裁剪平面以及视野。近和远裁剪平面的值取决于场景和物体的尺寸，所以我们为它们都设置了一个比较小的值。视野根据经验来设置。我们还要为阴影设置一个暗度值，Three.js 默认的 0.5 对这个应用来说很合适。随后，我们需要设置 Three.js 阴影贴图尺寸的大小属性。阴影贴图是一个额外的位图，专门用于渲染阴影的暗区域并最终和每个物体最终的渲染图像混合。我们将 SHADOW_MAP_WIDTH 和 SHADOW_MAP_HEIGHT 的值设为 2048，比 Three.js 默认的 512 要高。这将提供非常平滑的阴影；这个值越小，产生的阴影就会呈现越多的锯齿。尝试修改示例中的这个值来查看低分辨率的阴影贴图是如何影响阴影质量的。

最后，我们需要告诉 Three.js 哪些对象产生和接收阴影。Three.js 的网格默认既不产生也

不接收阴影，我们需要明确设置。在这个示例中，我们希望将三个几何体的阴影投到地板上，然后地板接收这些阴影，所以，我们将地板的 mesh.castShadow 属性设置为 false，mesh.receiveShadow 属性设置为 true；将立方体、球体、圆锥的 mesh.castShadow 属性设置为 true，mesh.receiveShadow 属性设置为 false。

最后的最后，我们希望阴影的强度随着聚光灯的光线强度变化而变化。然而，Three.js 的阴影贴图在渲染的时候并不会自动随光线强度变化而调整。实际上，它依赖于灯光的 shadowDarkness 属性。因此当灯光的颜色被用户改变的时候，我们需要手动去更新 shadowDarkness 的值。接下来的这段代码展示了 setShadowDarkness() 函数的实现，它基于光源 RGB 值的平均值来计算阴影的暗度。当你将聚光灯的颜色调暗时，会发现阴影也随着变淡了。

```
function setShadowDarkness(light, r, g, b)
{
    r /= 255;
    g /= 255;
    b /= 255;
    var avg = (r + g + b) / 3;

    light.shadowDarkness = avg * 0.5;
}
```

> 实时阴影是 WebGL 可视化体验中的一项神奇的增强技术，而 Three.js 可以帮助我们更方便地实现它。然而，它是有代价的。首先，阴影贴图是另一种类型的纹理贴图，它需要更多的显存。对于一个 2048 × 2048 的阴影贴图，我们需要额外的 4 MB 显存空间。试试看你是否能够用更小尺寸的阴影贴图来达到你想要的效果。而且，在某些图形设备上，离屏渲染阴影贴图可能会引入更多处理开销，从而显著降低帧率。因此你应当谨慎地使用这个特性。请务必做好足够的性能分析，并在需要的情况下降级到一个无需实时阴影的方案。

4.6 着色器

Three.js 内置了一套强大的材质集合。它们都是基于库中预置的 GLSL 着色器来实现的。这些着色器可以用来支持常见的着色需求，如 unlit、Phong 和 Lambert。但或许还存在很多别的需求。总的来说，材质应当可以实现无限多种类的效果，可以使用多种多样的属性，可以支持任意复杂的实现。举例来说，一个用于模拟风吹草动的着色器，可能需要支持调整草的高度和密度，以及调整风速和风向的属性。

随着计算机图形的发展，以及过去二十年相关领域产能价值的提升（从开始的为电影制作后期效果，到后来的运用于实时视频游戏），着色技术不再是艺术产品的尝试，而是成为了一个通用的程序问题。业界联合起来创建了一项可编程的管道技术，称为*可编程着色器*（programmable shader），而不是尝试预测每种可能的材质属性组合，并包含在运行时引擎的代码中。着色器允许开发者使用类 C 语言编写实现顶点级和像素级的复杂效果，并编译

成可以在 GPU 中运行的代码。使用可编程着色器，开发者可以创建出高性能、高度逼真的视觉效果，从预置的材质和灯光模型的限制中解放出来。

4.6.1 ShaderMaterial类：编写你自己的着色器代码

GL 着色语言（GLSL）是为 Open GL 和 OpenGL ES 开发的一种着色器语言（WebGL API 的基础）。GLSL 源代码通过 WebGL 上下文对象提供的方法来编译并在 WebGL 中执行，Three.js 隐藏了 GLSL 的底层细节，允许我们选择无需编写着色器的模式进行开发。对于多数的应用来说，预置的材质类型已经足够。但如果我们的应用需要一些预置类型中没有提供的效果，Three.js 也允许我们使用 `THREE.ShaderMaterial` 来进行定制化的着色器来发。

图 4-14 展示了 `ShaderMaterial` 应用的一个示例。这个示例可以在 Three.js 工程目录下的 examples/webgl_materials_shaders_fres nel.html 文件中找到，它演示了一个 Fresnel 着色器。Fresnel 着色器用于模拟光线穿过水和玻璃等透明介质时产生的反射和折射效果。



图 4-14：Fresnel 着色器提供了高度逼真的反射和折射效果

> Fresnel 着色器（读作“fre-nel”）以 Fresnel Effect（菲涅尔效应）命名。这个效应的最早记录出自法国物理学家奥古斯丁·简·菲涅尔（Augustin-Jean Fresnel，1788—1827）。菲涅尔通过研究光在不同介质中的传播，提出了波动理论。希望获取更多的信息，请访问 online 3D rendering glossary（http://www.3drender.com/glossary/fresneleffect.htm）。

示例中的设置代码按以下流程创建了一个 `ShaderMaterial` 对象：首先它复制了 `FresnelShader` 模板对象中的 uniform（参数）值——每个着色器实例都需要它自身的数据

副本——并将 GLSL 源码作为参数传给顶点和片段着色器。在初始化完成之后，Three.js 会自动处理着色器的编译和链接，并将 JavaScript 属性绑定到 uniform 的相应值上。

```
var shader = THREE.FresnelShader;
var uniforms = THREE.UniformsUtils.clone(shader.uniforms );

uniforms[ "tCube" ].value = textureCube;

var parameters = {
   fragmentShader: shader.fragmentShader,
   vertexShader: shader.vertexShader,
   uniforms: uniforms };

var material = new THREE.ShaderMaterial( parameters );
```

Fresnel 着色器的 GLSL 源代码如例 4-4 所示。这段源代码可以在 Three.js 工程目录下的 examples/js/shaders/FresnelShader.js 文件中找到。这个着色器代码由 Three.js 的活跃贡献者 Branislav Ulicny（更为人所知的是他的昵称 AlteredQualia）提供。让我们通读这段代码，来看看这个着色器是如何实现的。

例 4-4：为 Three.js 编写的 Fresnel 着色器

```
/**
 * @author alteredq / http://alteredqualia.com/
 * Based on Nvidia Cg tutorial
 */

THREE.FresnelShader = {

    uniforms: {

        "mRefractionRatio": { type: "f", value: 1.02 },
        "mFresnelBias": { type: "f", value: 0.1 },
        "mFresnelPower": { type: "f", value: 2.0 },
        "mFresnelScale": { type: "f", value: 1.0 },
        "tCube": { type: "t", value: null }

    },
```

`THREE.ShaderMaterial` 中的 `uniforms` 属性指定了 Three.js 在着色器被使用时会传给 WebGL 的值。回忆一下，着色器代码会对每个顶点和像素（片段）都执行一次。正如字面上的意思那样，着色器中的 uniform 属性的值不会随着顶点的切换而改变；它们本质上是作用于全部顶点和像素的恒定全局变量。这个示例中的 Fresnel 着色器定义了用于控制反射率和折射率的 uniform 属性（例如 `mRefractionRatio` 和 `mFresnelScale`）。它还为立方体纹理定义了一个 uniform 属性，作为整个场景的背景。类似于我们前面学习过的立方体环境映射，这个着色器也使用了将立方体映射像素渲染到表面来模拟反射的方式。然而使用这个着色器，我们不仅仅可以看到从立方体映射反射的像素，还可以观察到折射现象。

4.6.2　在Three.js中使用GLSL着色器代码

现在该来设置顶点和片段着色器了，首先是顶点着色器：

```
        vertexShader: [

            "uniform float mRefractionRatio;",
            "uniform float mFresnelBias;",
            "uniform float mFresnelScale;",
            "uniform float mFresnelPower;",

            "varying vec3 vReflect;",
            "varying vec3 vRefract[3];",
            "varying float vReflectionFactor;",

            "void main() {",

                "vec4 mvPosition = modelViewMatrix * vec4( position, 1.0 );",
                "vec4 worldPosition = modelMatrix * vec4( position, 1.0 );",

                "vec3 worldNormal = normalize( mat3( modelMatrix[0].xyz, ",
                "    modelMatrix[1].xyz, modelMatrix[2].xyz ) * normal );",

                "vec3 I = worldPosition.xyz - cameraPosition;",

                "vReflect = reflect( I, worldNormal );",
                "vRefract[0] = refract( normalize( I ), worldNormal, ",
                "    mRefractionRatio );",
                "vRefract[1] = refract( normalize( I ), worldNormal, ",
                "    mRefractionRatio * 0.99 );",
                "vRefract[2] = refract( normalize( I ), worldNormal, ",
                "    mRefractionRatio * 0.98 );",
                "vReflectionFactor = mFresnelBias + mFresnelScale * ",
                "    pow( 1.0 + dot( normalize( I ), worldNormal ), ",
                "    mFresnelPower );",

                "gl_Position = projectionMatrix * mvPosition;",
            "}"

        ].join("\n"),
```

顶点着色器代码是这个材质的主力。它使用相机位置以及模型（在这个示例中指的是用作表示气泡形状的球体）中每一个顶点的位置信息来计算出一个方向向量，用于计算每个顶点的反射和折射系数。注意在顶点和片段着色器中的 `varying` 声明。与 uniform 类型的变量不同，varying 类型的变量在每个节点中都会被计算，然后从顶点着色器传递到片段着色器。通过这种方式，顶点着色器可以输出数值到内置的 `gl_Position` 变量中，而这是它主要的工作。对 Fresnel 着色器来说，varying 输出的是反射和折射系数。

Fresnel 顶点着色器还使用了许多我们在这里没有看见的 varying 和 uniform 变量：`modelMatrix`、`modelViewMatrix`、`projectionMatrix` 和 `cameraPosition`。这些变量是由 Three.js 预先定义的，并自动传递到 GLSL 编译器。这些值不需要，或者说不应该由着色器的编写者显式声明。

- `modelMatrix`（uniform）

  模型（网格）的世界变换矩阵。正如在 4.2 节中所讨论的那样，这个矩阵在每一帧都会被 Three.js 计算出来，用于表示物体在世界空间中的位置信息。在这个着色器中，它被

用来计算每个顶点在世界空间中的位置。

- `modelViewMatrix`（uniform）

  用于表示每个物体在相机空间中的位置变换信息，也就是在坐标系中相对于相机的位置和方向，这对于计算和相机相关的值（例如在这个着色器中用于确定反射和折射的值）特别方便。

- `projectionMatrix`（uniform）

  用于计算我们熟悉的从相机空间到屏幕空间的 3D 到 2D 的映射。

- `cameraPosition`（uniform）

  由 Three.js 维护，表示相机在世界空间中的位置，自动传入。

- `position`（varying）

  模型空间中的顶点位置。

- `normal`（varying）

  模型空间中的法向量信息。

这个顶点着色器还使用了内置的 GLSL 函数 `reflect()` 和 `refract()`，用来基于相机方向、法向量和折射率来计算反射和折射向量信息。（由于这些函数在进行如 Fresnel 方程这类光线计算时非常有用，所以被作为 GLSL 语言的内置函数。）

最后，注意 `Array.join()` 函数的使用，它被用来将顶点着色器的代码字符串拼在一起。这展示了另一种拼接使用 GLSL 编写的着色器代码的长字符串的有用方法。我们使用 `join()` 函数来在代码中插入新行，而不是在每一行的结尾处换行并添加字符串连接符。

从这里起，片段着色器的工作就非常简单了。它使用顶点着色器计算出的反射率和折射率的值，来索引到通过 uniform 变量 `tCube` 传入的立方体纹理上的颜色值。这个变量是 `samplerCube` 类型的，`samplerCube` 类型是一种用于处理立方体纹理的 GLSL 类型。我们使用 GLSL 函数 `mix()` 来混合两种颜色，构建出最终的像素信息并将其输出存储在内置的 `gl_FragColor` 变量中。

```
fragmentShader: [

    "uniform samplerCube tCube;",

    "varying vec3 vReflect;",
    "varying vec3 vRefract[3];",
    "varying float vReflectionFactor;",

    "void main() {",

        "vec4 reflectedColor = textureCube( tCube, ",
        "    vec3( -vReflect.x, vReflect.yz ) );",
        "vec4 refractedColor = vec4( 1.0 );",

        "refractedColor.r = textureCube( tCube, ",
        "    vec3( -vRefract[0].x, vRefract[0].yz ) ).r;",
        "refractedColor.g = textureCube( tCube, ",
```

```
            "    vec3( -vRefract[1].x, vRefract[1].yz ) ).g;",
            "refractedColor.b = textureCube( tCube, ",
            "    vec3( -vRefract[2].x, vRefract[2].yz ) ).b;",

            "gl_FragColor = mix( refractedColor, ",
            "    reflectedColor, clamp( vReflectionFactor, ",
            "    0.0, 1.0 ) );",

        "}"

    ].join("\n")

};
```

创建定制化的着色器看起来需要很多工作，但这是值得的，因为它输出了一个非常接近真实世界的光学模拟效果。而 Three.js 帮忙完成了其他的机械工作——更新每个物体的世界矩阵，追踪相机，预定义许多 GLSL 变量，编译和链接着色器代码——为我们省下了大量的开发和调试时间，使得着色器的开发工作变得方便且吸引人。有了这个框架，你可以轻松尝试编写你自己的着色器。我建议从 Three.js 示例中提供的 Fresnel 和其他着色器开始。Three.js 提供了多种不同类型的特效，通过这些示例可以学到非常多的东西。

4.7 渲染

这一章我们使用了 Three.js 中的很多方法来不断提升真实度，从简单的几何形状到材质、纹理、灯光和阴影，最终使用 GLSL 编写了我们自己的着色器。每一步，我们都比上一步创造了更具真实感的图形，但我们还差最后一步：渲染。

Three.js 3D 场景控制的最终输出是渲染在浏览器 Canvas 元素上的 2D 图像。无论是使用 WebGL，还是 2D Canvas 绘图 API，抑或是通过更改 CSS 来使得元素在页面上移动，都无关紧要；我们的最终目的都是绘制像素点。我们选择 WebGL，只是因为它具有高性能。使用其他技术，我们或许能实现同样的视觉效果，但性能无法接受。所以我们经常选择 WebGL。

也就是说，即便使用了 WebGL，我们在具体渲染图像的时候也有很多选择。例如，API 允许我们选择是否使用*深度缓冲*（Z-buffering）渲染（这是一种通过在硬件中使用额外的存储空间来存储物体的深度信息，以实现只绘制场景中最前面像素的技术）。这是可选的。如果我们没有启用深度缓冲，我们的应用则需要自己处理物体的排序，这可能需要深入到面片级别。这听起来很麻烦，但在某些应用场景下，我们或许恰恰需要这么做。这仅仅是我们对于渲染的一个可选项。

Three.js 的设计初衷是使得基础的图形处理变得更加简单。使用内建的 WebGL 渲染器，开发者无需花费太多精力就可以创建出游戏级别的图形效果。正如我们在前面的示例中所看到的那样，这可以通过以下简单的步骤来实现：创建渲染器，设置视口尺寸，以及调用 `render()` 函数。但 Three.js 允许我们做更多的事情，它提供了能够控制 WebGL 底层渲染流程的能力。通过将这些能力结合高级的渲染技术，例如后处理、多道渲染以及延迟渲染，我们可以创建高度仿真的效果。

4.7.1 后处理和多道渲染

有时候，一次渲染并不够。一个场景经常需要通过多次渲染来创建高质量的、真实的图像。这些独立的渲染最终在一个被称为*多道渲染*（multipass rendering）的流程中合成到一起，产生最终的图像。许多多道渲染方法使用*后处理*（post-processing）技术，或者通过图像处理技术来提升图像质量。

后处理和多道渲染在实时 3D 渲染中越来越流行，因此 Three.js 的作者花了很大的工夫来支持它。图 4-15 展示了一个精细而生动的示例，它使用 Three.js 后处理编写，作者是 AlteredQualia。加载 examples/webgl_terrain_dynamic.html 文件。壮观的鸟群在雾霭的晨光中飞越奇幻的地表。单纯基于噪声程序生成的地形并不足以令人印象深刻，该场景特别使用了多道渲染工序，包括使用 bloom 着色法来强调阳光在晨雾中的漫射效果，以及使用高斯滤镜让场景产生轻微模糊来增强场景的宁静氛围。



图 4-15：动态地形程序示例，使用多道后处理工序渲染。程序由 AlteredQualia 提供；鸟儿由 Mirada 提供（来自 RO.ME fame）

Three.js 的后处理依赖于以下特性。

- 通过 `THREE.WebGLRenderTarget` 对象支持*多重渲染目标*（multiple render target）。通过多重渲染目标，场景可以被多次渲染到离屏位图上，并合成到最终的图像中。（源文件：src/renderers/WebGLRenderTarget.js。）
- `THREE.EffectComposer` 类实现了一个多道渲染循环。这个对象包含一个或多个*渲染工序*（render pass）对象，它们会被依次调用来进行场景渲染。每一道工序都拥有访问整个场景以及上一道工序处理产生的图像数据的权限，从而进一步细化图像。

`THREE.EffectComposer`，以及使用它来实现多道渲染的示例，可以在 Three.js 工程目录中的 examples/js/postprocessing/ 和 examples/js/shaders/ 文件夹中找到。浏览这些目录，你将会发

现一个后处理特效的宝库。

4.7.2 延迟渲染

我们再来看看另一种渲染方法：*延迟渲染*（deferred rendering）。正如其名，这种方法直到通过不同的来源计算出最终的图像结果，才把这个结果渲染到 WebGL canvas 上。与多道渲染持续渲染一个场景，不断改进图像，最终复制到 WebGL canvas 中不同，延迟渲染在初始阶段使用了多个缓冲（就是纹理贴图）来收集着色器的计算结果数据，并在接下来的阶段中，使用初始阶段收集的结果来计算像素值，这种方式非常耗费存储空间和计算能力，但它也可以创造高度真实的效果，尤其是对于光线和阴影的处理。参见图 4-16。

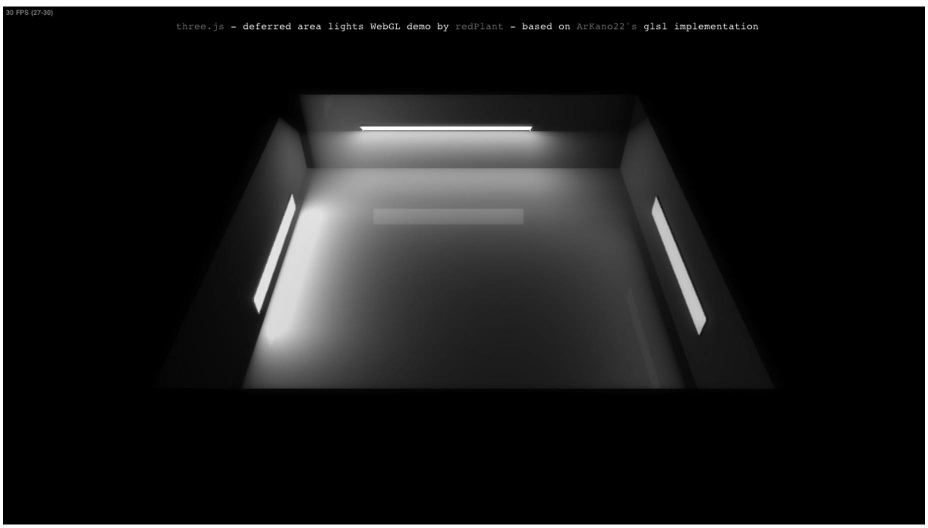


图 4-16：使用延迟渲染的逐像素光线处理（另见彩插图 4-16）

4.8 小结

本章覆盖的内容非常广，涉及了 Three.js 提供的大部分绘图和渲染能力。我们了解了如何使用预置的几何形状类来创建 3D 立方体、网格和参数化的挤出图形。我们讨论了 Three.js 中用于组织复杂场景的场景图和空间变换层级结构。我们亲手实践了材质、纹理和灯光。最后，我们了解了可编程着色器和高级的渲染技术，如后处理和延迟渲染是如何提升可视化的真实程度的。Three.js 将强大的图形处理特性以简单易用的库的形式提供出来。这些能力结合 WebGL 的原生能力，使得我们几乎能够创建出我们能够想到的全部 3D 可视化效果。

第5章

3D动画

动画意味着随着时间改变屏幕上的图像。有了动画以后，静态的 3D 场景就有了生命。尽管有很多可用于动画的技术以及从概念上对问题建模的方式，最终，动画要解决的事只有一件：让像素移动。

WebGL 本身并没有对动画的原生支持。尽管如此，WebGL API 的能力和速度允许我们以高达 60 帧每秒的频率渲染复杂的图形并修改它们，并允许我们以多种方式实现 3D 动画内容。结合现代浏览器对运行时架构的改进，这些动画可以与页面上的其他元素无缝融合。

动画可以用于改变 WebGL 场景中的任意元素：变换、几何形状、纹理、材质、灯光和相机。物体可以移动、旋转、伸缩或沿轨迹运动；几何形状可以弯曲、扭曲或者变成其他形状；纹理可以移动、缩放、旋转和滚动，或者随着帧变化修改其像素点；材质颜色、镜面高光、透明度等都可以随着时间变化；灯光可以闪烁、移动以及改变自身颜色；相机可以移动、旋转来创建类似电影的效果。动画有无限的可能。

在本章中，我们会研究多种动画技术，以及用于实现它们的工具和库。这些技术基于电影和电子游戏行业多年的实践，并由严格的数学论证支持。WebGL 中的动画是一个发展中的领域，所以我们的探索涉及多种不同方案的混合使用。Three.js 提供了能够很好地处理部分使用场景的动画工具。我们也会研究另一个开源库——Tween.js。Tween.js 是一个小巧易用的、用于创建简单过渡的库。然而这些还远远不足，如果你的应用十分复杂，那你可能需要创建你自己的动画引擎。

创建 WebGL 动画内容涉及以下一个或多个概念，这些概念会在本章中进行详细介绍。

- 使用 `requestAnimationFrame()` 来驱动运行循环。
- 在每次运行循环执行的时候，以编程方式改变可视物体的属性。这适用于创建简单的动画，如使一个物体绕单个坐标轴旋转。在一个物体的位置、方向或其他属性可以表示为

关于某个变量（例如时间）的函数时，这项技术也非常有用。总的来说，这是最简单的动画技术实现，但它仅限于非常有限的使用场景。

- 使用*补间*（tween）来实现属性从一个值到另一个值的平滑变化。补间非常适合用来实现一次性的简单效果（例如，将一个物体从一个位置沿直线移动到另一个位置）。
- 使用*关键帧*（keyframe），它的数据结构表示了一系列沿时间轴排列的离散值，并包含一个用于生成平滑效果的*插值*（interpolate）计算引擎。关键帧适用于平移、旋转、缩放以及材质颜色等简单的属性的基础动画。关键帧允许我们在一次动画中创建一系列的过渡，而不是像补间那样只支持从一个值到另一个值的单次过渡。
- 沿*路径*（path，用户定义的曲线或线段）移动物体，基于公式或预定义的路径数据来创建复杂的有机运动效果。
- 使用*变形目标*（morph target），通过整合一系列相互独立的形状，实现对几何形状的变形。这是一项高级的用于表现面部表情或非常简单的人物动画的技术。
- 使用*蒙皮*（skinning）来基于运动的骨骼对几何形状进行变换。这是实现人物动画和其他复杂形状动画的优先选择。
- 使用*着色器*（shader）来随着时间改变顶点位置和像素值。某些时候，一个所需的动画效果最好基于顶点级或像素级进行计算，这些计算最好使用 GLSL 来实现。着色器同时可以用于其他技术的高性能实现，特别是变形和蒙皮，如果用 CPU 来处理，会非常耗费计算。

在一个应用中，通常会使用这些方法中的一种以上，或者是全部。对于每项技术应该应用于哪个具体的场景，并没有一个硬性的规定，尽管其中某些对于特定的场景会更加合适。通常技术的选择取决于产品考虑；例如，如果你的团队成员中没有设计师，那么由一个程序员来使用代码生成动画也许会更容易。其他时候，它取决于个人偏好。3D 动画是艺术和科学的组合，也是生产与工程的结合。

5.1 使用requestAnimationFrame()来驱动动画

在前面的章节中我们已经了解到如何使用 `requestAnimationFrame()` 来支持应用的运行循环，这是一个 Web 浏览器所支持的比较新的 API。

`requestAnimationFrame()` 的设计目的是，让通过 JavaScript 代码驱动的 Web 应用能够有一致、可靠的视觉效果。包括 DOM 元素变化、布局调整、使用 CSS 的样式修改，或基于 Canvas 和 WebGL 这样的绘图 API 创建的图形。`requestAnimationFrame()` 最早出现在 Firefox 4 中，并最终被全部浏览器所接受。Mozilla 的 Robert O'Callahan 原先是为了寻找一种保证浏览器内建效果（如 SVG 或 CSS 过渡）与用户的 JavaScript 代码同步的方法。

从前，Web 应用使用*定时器*（timer）来控制页面内容的动画，通过 `setTimeout()` 或 `setInterval()` 这两个函数。随着 Web 应用开始包含更复杂的动画和交互，这种方法明显遭遇了以下一些关键问题。

- 定时器以设置好的恒定（或尽可能接近）间隔来调用回调函数，无论是否是绘制图形的最佳时机。
- 在定时器回调中执行的 JavaScript 代码无法与其他浏览器计算产生的动画时间轴同步（如

SVG 或 CSS 过渡）。

- 无论页面或标签页是否可见，或浏览器窗口是否被最小化，定时器都会被重复执行，这会造成潜在的绘图资源浪费。
- JavaScript 应用代码不知道显示器的刷新率，所以它只能主观地设定计时器间值。例如将其设置为 1/24 秒，那么使用 60 Hz 刷新率显示器的用户将无法感受到流畅的视觉效果；如果你将这个值设置为 1/60 秒，那么在刷新率较低的显示器上，你就耗费了额外的 CPU 资源来绘制永远不会被显示的内容。

`requestAnimationFrame()` 被设计来解决以上所有问题。回忆一下前几章的例子，我们的运行循环类似于以下这种形式：

```
function run() {

    // 请求下一帧动画
     requestAnimationFrame(run);

    // 启动动画
    animate();

    // 渲染场景
    renderer.render( scene, camera );
```

注意，调用 `requestAnimationFrame()` 时没有传入时间值。我们没有指定浏览器在某个具体时间或间隔时调用我们的动画和绘制代码，我们只告诉它在*页面再次呈现的时候调用*。这是和计时器的关键区别。使用了这种方式，浏览器就能在它内部重绘周期的时候调用用户代码。这有几个好处。第一，浏览器可以按照所需进行频繁或不频繁的调用。当浏览器还有足够的空闲时间时，它可以试着提高帧率来赶上显示器的刷新率。相反，如果页面或标签隐藏了，或整个浏览器最小化了，它可以减少回调函数的执行次数，优化电脑或设备的资源使用。第二，浏览器可以批量处理用户绘图操作，从而减少绘制屏幕的次数，节省资源。第三，在 `requestAnimationFrame()` 中执行的所有绘制代码最终会与其他绘图操作（包括浏览器内部的）进行合成（composite）。最终产生更平滑、更快速、更高效的页面绘制及动画。

5.1.1 在你的应用中使用requestAnimationFrame()

正如最近许多基于 HTML5 特性的开发一样，`requestAnimationFrame()` 不一定会被市面上全部浏览器的所有版本支持，尽管支持它的浏览器已经越来越多。同时，在它从一个浏览器的实验性功能真正发展到 W3C 建议的稳定版本的过程中，各浏览器早期实现的都是这个函数的带私有前缀版本。幸好我们可以使用一个由 Google 工程师 Paul Irish 创建的 polyfill。代码如例 5-1 所示，你也可以在本书的示例文件 libs/requestAnimationFrame/RequestAnimationFrame.js 中找到这段代码。它会尝试在所有浏览器版本中寻找这个函数的正确名称实现，如果找不到，则自动降级到每秒 60 帧的 `setTimeout()` 方案。

例 5-1：RequestAnimationFrame polyfill，由 Paul Irish 提供

```
/**
 * 以跨浏览器方式提供requestAnimationFrame
```

```
 * http://paulirish.com/2011/requestanimationframe-for-smart-animating/
 */

if ( !window.requestAnimationFrame ) {

    window.requestAnimationFrame = ( function() {

        return window.webkitRequestAnimationFrame ||
        window.mozRequestAnimationFrame ||
        window.oRequestAnimationFrame ||
        window.msRequestAnimationFrame ||
        function( /* function FrameRequestCallback */ callback,
            /* DOMElement Element */ element ) {

            window.setTimeout( callback, 1000 / 60 );
        };

    } )();

}
```

> 有些读者或许不太了解 polyfill 这个单词，它是指用代码（通常是 JavaScript）来实现浏览器中没有提供的功能。polyfill 通常用在不支持新功能或实验性功能的老版本浏览器中。这个单词由英国工程师 Remy Sharp 提出。如果想了解更多关于这个单词的背景及来源，请浏览 Sharp 的博客文章（http://remysharp.com/2010/10/08/what-is-a-polyfill/）。

`requestAnimationFrame()` 成功运用的一个关键是确认你在执行其他用户代码之前就请求了下一帧，正如早先我们在展示过的运行循环代码片段中所做的那样。这对异常处理非常重要。如果你在动画回调函数中驱动你的整个 3D 应用，而代码在请求下一帧之前产生了一个异常，你的整个应用就会挂掉。然而如果你在做其他事情之前先请求了下一帧，那么至少还可以保证程序的持续运行。即使其他部分产生了错误，应用的一部分也还能够工作并重绘元素。

5.1.2 requestAnimationFrame()和性能

尽管 `requestAnimationFrame()` 对动画性能有利，但也需要小心使用。如果浏览器每秒 60 次调用你的回调，你需要保证这个回调只需花费 16 毫秒或更少的时间，否则用户会觉得你的应用失去了响应。因为 16 毫秒很短，所以你必须尽可能减少操作，只做必需的绘图修改。优秀的 3D 应用或许需要考虑结合使用定时器、worker 以及 CSS 变换与过渡等动画技术，来实现尽可能灵敏、强大及低资源占用的效果。

> `requestAnimationFrame()` 可以说是 HTML5 引进的一个最重要的特性。关于这个话题，本节只做了非常粗略的探讨。有很多优秀的在线资源可供我们更好地了解它。在网络上使用这个函数名来搜索，你会找到大量相关的文章、背景资料、引导文章、提示和诀窍，以及对其底层实现的解释。

5.1.3　基于帧的动画和基于时间的动画

早期计算机动画系统效仿了之前电影中的技术，连续在屏幕上展现静态图像（如果是在基于矢量的图形中，那就是程序生成的一系列矢量图）。每个这样的图片被称为一帧（frame）。历史上，电影是以每秒 24 张图片的速度拍摄和放映的，所以帧率（frame rate）为每秒 24 帧。这个速度在低亮度下的电影屏幕中足够了。但是，在计算机生成动画及 3D 游戏的世界，我们能够察觉和体会到更高分辨率的存在，高到 30 帧每秒、60 帧每秒或更高。尽管如此，许多动画系统，例如 Adobe Flash，最开始采用了 24 帧每秒的速度，因为传统的动画开发者已经习惯了。目前，这个帧率变化了，Flash 可以最高支持 60 帧每秒，但离散帧的概念依然在。这种将动画组织为一系列离散帧的做法被称为*基于帧的动画*（frame-based animation）。

基于帧的动画有一个严重的缺点：因为固定了具体的帧率，所以动画师可以确保动画不会比预先设置的帧率高，即便电脑可以支持更高的帧率。这对电影来说没有问题，因为相关硬件在整个业界都是一致的。然而，在计算机动画中，性能会因为设备的变化有很大差别。如果你创建的是 24 帧每秒的动画，但你电脑屏幕的刷新率为 60 Hz，你就失去了原本可以看到的更多细节，以及更流畅的播放体验。

另一种被称为*基于时间的动画*（time-based animation）的技术解决了这个问题。在基于时间的动画中，一系列矢量图形连接到特定的时间点上，而不是在已知帧率下的某些具体帧。因此，计算机可以显示这些图片，并在其中加入插值，尽可能高频地显示最好的图片效果及最平滑的过渡。在之前章节的例子中，我们使用了基于时间的动画，在每次运行循环中，`animate()` 函数计算了当前帧和前一帧之间的时间差，并使用它去计算旋转角度。本章及接下来几章的所有例子都使用基于时间的动画。所以尽管帧（frame）这个单词在 `requestAnimationFrame()` 名称的最后，但它同样适用于基于时间的动画。

5.2　使用程序更新属性的方式来构建动画

到目前为止，在 WebGL 场景中启动一个动画的最简单方式是编写代码，在运行循环每次执行的时候更新某个物体的属性值。我们在前面的章节中已经看过相应的示例。我们通过更改立方体的 `rotation.y` 属性来旋转第 3 章中的 Three.js 立方体。也就是说，在每一帧中我们动态地修改了物体绕 y 轴的旋转角度。让我们重温一下这段代码：

```
var duration = 5000; // ms
var currentTime = Date.now();
function animate() {

    var now = Date.now();
    var deltat = now - currentTime;
    currentTime = now;
    var fract = deltat / duration;
    var angle = Math.PI * 2 * fract;
    cube.rotation.y += angle;
}
```

变量 `duration`、`currentTime`、`now` 和 `deltat` 用于计算基于时间的旋转动画相关值。在这

个示例中，我们期望立方体以五秒钟旋转一周的速率绕 y 轴旋转。每次计算出来的角度值（`angle` 变量）是整个旋转过程的一个分量，这个量会被累加到立方体当前的 `rotation.y` 属性上。回想一下，旋转在 Three.js 中是以弧度，即单位圆上的距离的形式来表示的；也就是说，`Math.PI * 2` 等于旋转一周（360 度）。

这个概念可以延伸到给场景中的任意东西添加动画：位置、旋转、缩放、材质颜色、透明度，等等。除此之外，它实际上是完全通用的：通过使用 JavaScript 代码来更新属性，我们可以进行任意的计算更新。数学公式、布尔逻辑、统计值、数据流、实时传感器输入等都可以用来驱动动画。因此这对科学绘图和数据可视化领域来说是一项非常有用的技术：它可以用于描绘太阳系、物理过程和自然现象，也可以用来展现时序信息、统计分析、地理数据、网站流量以及其他动态的数据驱动的信息。它在音乐可视化这类活泼的娱乐性应用的创建中也有出色表现。

图 5-1 描绘了 Ellie Goulding's Lights 的广阔世界，这是一个由英国的交互设计代理公司 Hello Enjoy 开发的 WebGL 音乐可视化项目。这个项目已经发布了相当一段时间，但它依然可以算作一个非常有冲击力的作品。灯光交替明灭，彗星尾灯在场景中描绘着曲线，彩色的圆球随颜色变换忽隐忽现，探照灯疯狂地旋转，水滴形的气球从五彩起伏的地面上绽放——所有这些都紧随着音乐的节拍。这是养眼的视觉盛宴，并且所有的效果都是由程序计算出来的。

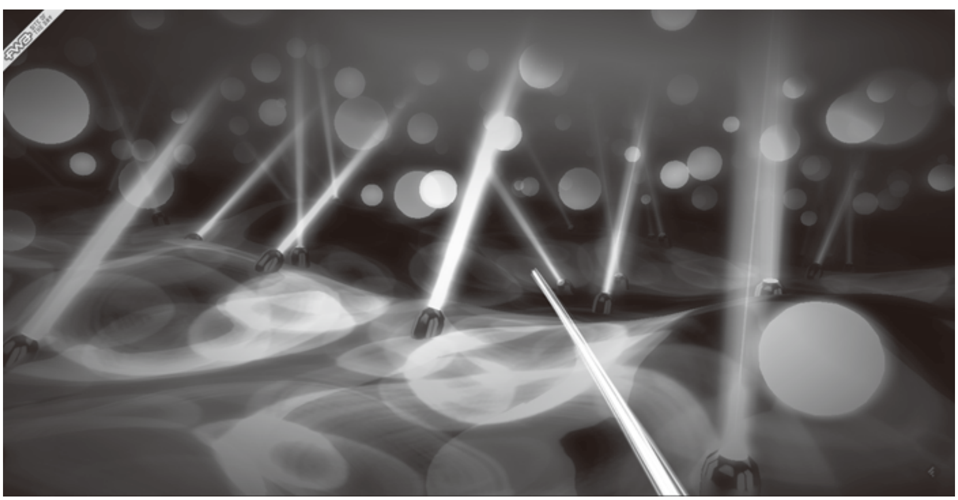


图 5-1：Ellie Goulding's Lights——一个用程序动画创建的音乐可视化项目；图片由 Hello Enjoy 提供（另见彩插图 5-1）

例 5-2 展示了这个可视化效果动画的一部分代码。这个应用的 `update()` 方法在每次运行循环执行的时候都被调用。应用依次对场景中的每个对象调用 `update()` 方法。下面的代码片段来自 `LIGHTS.StarManager.update()` 方法，这个方法用于控制背景星星的动画。星星被渲染为一个 `THREE.ParticleSystem` 对象中的粒子群。用粗体标出的代码行展示了每个星星的 RGB 颜色值是如何随着时间而变化的，以及通过一个衰减因子和取余运算符（`%`）来创造

出闪烁的效果。

例 5-2：节拍动画——Ellie Goulding's Lights 的代码片段

```
update: function() {

    var stars = this.stars,
        deltaTime = LIGHTS.deltaTime,
        star, brightness, i, il;

    for( i = 0, il = stars.length; i < il; i++ ) {

        star = this.stars[ i ];

        star.life += deltaTime;

        brightness = (star.life * 2) % 2;

        if( brightness > 1 )
            brightness = 1 - (brightness - 1);

        star.color.r =
        star.color.g =
        star.color.b = (Math.sin( brightness * rad90 - rad90 ) + 1) * 4;
    }

    this.particles.__dirtyColors = true;
},
```

尽管以编程方式实现的动画非常灵活和强大，但它的局限性仍然不容忽视。它需要手动为每个效果编写代码，因此如果要进行多种不同类型物体的动画，将会非常吃力。同样，相比其他数据驱动的方法，如补间和关键帧（稍后将会讲述这些概念），它也更为复杂。最后，它将程序员而不是艺术家摆在了实现效果的关键位置，而创作理想的视觉效果这项工作或许更适合艺术家来做。不管怎么说，编程实现的动画依然是一个为场景添加灵动效果的、快速而简单的优秀方案，实现的效果也可以十分惊人，正如 Ellie Goulding's Lights 中那样。

5.3 使用补间来进行动画过渡

许多动画效果比起用程序在每次运行循环中计算更新，更适合用一个数据结构来表示。应用提供一系列值和时序，以及一个用于计算每帧属性值的引擎。这种数据驱动的方法称为*补间*（tweening）。

补间指的是计算一对值中间插入的其他数值的过程。有了补间，动画绘制者只需提供动画的起始值和结束值，用于动画时间过程中的中间值（补间）由引擎自动计算生成。补间是实现由一个状态切换到另一个状态的一次性简单动画的完美选择，如点击鼠标移动一个物体。

5.3.1 插值

补间通过一项被称为*插值*（interpolation）的数学技术来实现。插值指的是基于一个标量输入（如时间或分数值）来计算两个值之间的一个值。图 5-2 描绘了插值。对于任意值 `A` 和

B，以及一个 0 和 1 之间的分数 u，插值 P 可以通过公式 A + u * (B – A) 计算出来。对于图 5-2 中的示例，我们可以看到插值 P(u) = 0.4。这是最简单的、被称为线性插值（linear interpolation）的插值方法，因为用来计算插值结果的函数图像可以描绘为一条直线。其他更复杂的插值函数，如样条（一类曲线）和多项式，同样被广泛用于动画系统中。我们稍后将会了解基于样条的动画。

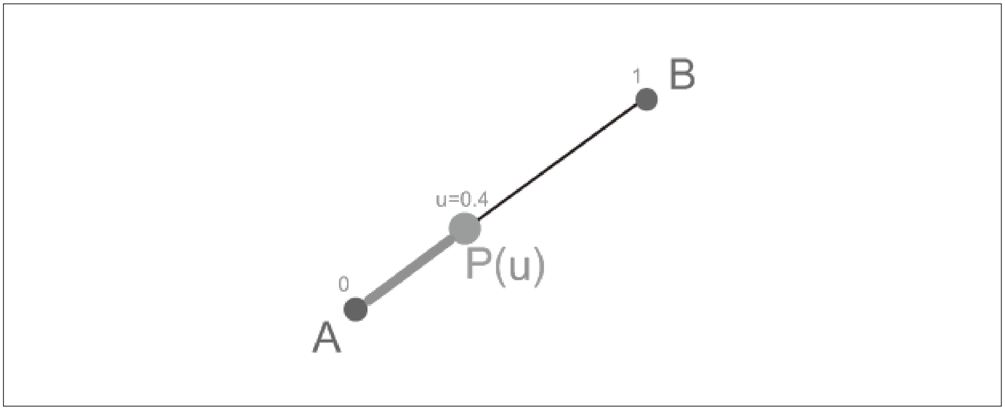


图 5-2：线性插值；转载已被许可（另见彩插图 5-2）

插值被用来计算 3D 位置、旋转、颜色、标量值（例如透明度）等的补间。对于一个包含多个组成部分的值，例如 3D 向量，线性插值补间通过计算每一个分量的补间而得到。例如，对于从点 A (0, 0, 0) 到点 B (1, 2, 3) 的 3D 向量 AB，基于 u = 0.5 的插值 P 为 (0.5, 1, 1.5)。

5.3.2 Tween.js库

自行实现简单的补间是非常容易的事。尽管如此，当你需要非线性的插值函数，或者其他像淡入 / 淡出（动画加速进入，减速退出）这类花哨的效果时，问题就变得复杂了。你需要一个现成的库，而非构造自己的补间体系。Tween.js 是一个流行的开源补间通用库，作者是 Soledad Penadés。它被应用在许多流行的 WebGL 工程中，包括 RO.ME、WebGL Globe 以及 Mine3D——经典单人游戏 *Minesweeper* 的 Web 版。

文件 Chapter 5/tweenjstweens.html 中的示例包含一个用于测试 Tween.js 各项选项的沙盒。图 5-3 是示例的截图。沙盒利用 Tween.js 为一个表面贴图的立方体添加各种变换：位置、旋转、材质颜色以及透明度。沙盒界面提供了用于调整补间时长和延时（补间开始之前停顿的时间）的滑块、用于启用 / 禁用各种补间的复选框，以及一个用于勾选补间是否循环（持续重复）的选项。此外还有一个用于控制缓动函数的选项，我们将在下一节予以说明。你可以调整不同的选项来看看它们是如何改变效果的。

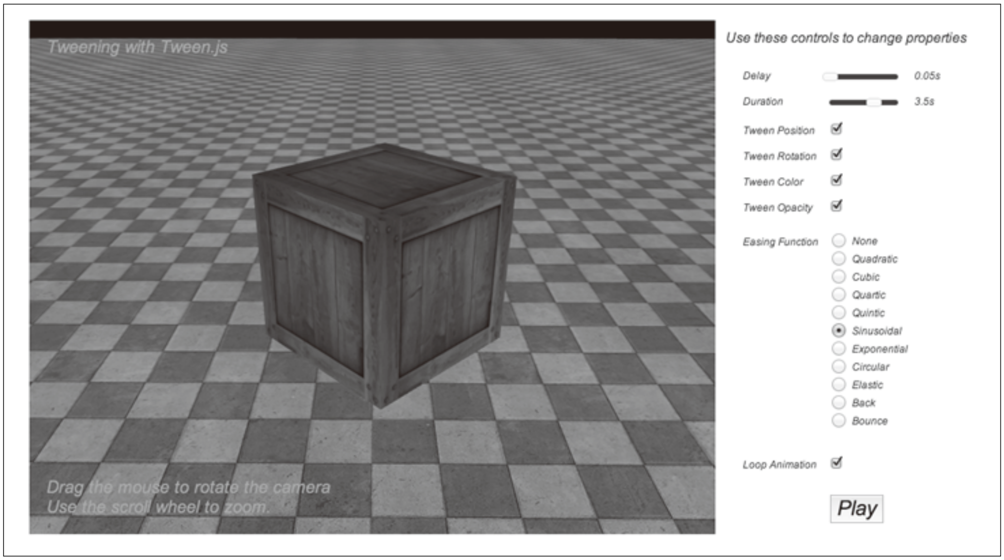


图 5-3：利用 Tween.js 控制过渡动画

Tween.js 非常易用。它的语法非常简单，并且多亏 JavaScript 语言本身的多态性，我们可以利用基本相同的函数调用方式对任何属性进行修改。它还使用了类似 jQuery 的链式调用语法，使得整个表达式看起来十分简洁。让我们来看看例 5-3，它展示了 `playAnimations()` 函数代码的一部分，每当一个属性发生变化时，这个函数就被调用来触发一个补间。

例 5-3：用于产生位置动画的 Tween.js 代码

```
positionTween =

    new TWEEN.Tween( group.position )
     .to({ x: 2, y: 2, z:-3 }, duration * 1000)
    .interpolation(interpolationType)
    .delay( delayTime * 1000 )
    .easing(easingFunction)
    .repeat(repeatCount)
    .start();
```

这个位置补间由一个方法链构成。

- 构造器 `new TWEEN.Tween`。它需要一个参数，这个参数表示包含需要进行补间的属性的对象。
- `to()`。它需要一个定义补间目标属性值的 JavaScript 对象，以及以毫秒为单位定义的补间时长。
- `interpolation()`，一个用于定义插值类型的可选方法。它的默认值是线性插值（`TWEEN.Interpolation.Linear`），因此在使用线性插值的时候，这个调用可以省略。
- `delay()`，一个用于在补间开始之前插入延迟的可选方法。
- `easing()`，一个用于定义缓动函数（在下一节说明）的可选方法。
- `repeat()`，一个用于定义补间重复次数的可选方法（默认为 0）。

- start()，开始补间。

注意这些方法也可以分开调用。每个补间——位置、旋转、材质颜色以及透明度——都可以通过类似的方式设置。Tween.js 的强大之处在于你只需向 to() 方法提供产生变化的属性值，而无需对象的所有属性值。例如一个仅改变绕 *y* 轴的旋转值的旋转补间，可以通过如下形式来构建：

```
rotationTween =

    new TWEEN.Tween( group.rotation )
     .to( { y: Math.PI * 2 }, duration * 1000)
    .interpolation(interpolationType)
    .delay( delayTime * 1000 )
    .easing(easingFunction)
    .repeat(repeatCount)
    .start();
```

补间被创建并启动之后，我们需要确保 Tween.js 在每个动画帧中都会更新。这一步由应用本身负责执行，因此我们在 run() 函数中添加了下面这行代码：

```
TWEEN.update();
```

Tween.js 在内部维护了一个全部运行中补间对象的列表，并依次调用它们的 update() 方法。update() 计算经过的时间，并根据缓动函数、延时和重复选项，最终计算出在 to() 方法中指定的属性值的当前值，并将其赋予相关属性。这是一个美丽、优雅而简单的，用于创建属性变化的方案。使用这个方案，我们无需再在每一帧中手动更新属性值。

5.3.3 缓动

由于物体始终进行匀速变化，使用线性插值的基础补间会产生一个僵硬而不自然的效果。自然界中的物体运动与此完全不同，它们的运动带有惯性、动量、加速度，等等。有了 Tween.js，我们就可以利用内置的*缓动*（easing）——应用于补间开始和结束阶段的非线性函数——来创建更自然的补间。缓动是使得你的补间看起来更为真实的优秀工具。它还可以用于在应用中没有整合物理引擎的情况下，近似地模拟一些物理效果。

尝试补间沙盒中的不同缓动函数，并注意它们的效果。正如它们的名字那样，多项式缓动函数 Quadratic、Cubic、Quartic 和 Quintic 分别表示通过二次、三次、四次和五次函数来实现缓动效果。而其他缓动函数提供了正弦波、弹跳以及弹簧效应。每个缓动函数均可用于缓动进入（补间的开始阶段）、缓动退出（补间的结束阶段），或者两者都包括。

缓动函数实际进行的操作是修改时间的值。例 5-4 展示了缓动函数 TWEEN.Easing.Cubic 的代码。这个缓动函数的输入是一个 [0..1] 值域上的数值（例如整个补间时长的一个分量）。这里的输入 k 在缓动函数中被进行了三次幂处理。也就是说，输入的 k 值越小，返回值就会更小。不过，当 k 的值达到 1 时，返回值也是 1。

例 5-4：Tween.js cubic 缓动函数

```
Cubic: {

    In: function ( k ) {
```

```
            return k * k * k;

        },

        Out: function ( k ) {

            return --k * k * k + 1;

        },

        InOut: function ( k ) {

            if ( ( k *= 2 ) < 1 ) return 0.5 * k * k * k;
            return 0.5 * ( ( k -= 2 ) * k * k + 2 );

        }

    },
```

Tween.js 的缓动函数基于 Robert Penner（http://www.robertpenner.com/index2.html）在动画方面的精细成果。它们提供了广泛而强大的缓动方程，包括线性方程、二次方程、四次方程、正弦方程和指数方程。Penner 的成果被从原始的 ActionScript 版本转换为各种语言的版本，包括 JavaScript、Java、CSS、C++ 和 C#，并被整合进 jQuery 的动画通用组件中。

正如我们方才看到的那样，补间非常适合用于创建简单并且看起来自然的效果。Tween.js 还允许你将不同的动画链接起来组成更复杂的动画。尽管如此，当你开始创建复杂的动画队列的时候，一定希望能够有一个更通用的解决方案。这时候，关键帧动画就派上用场了。

5.4 使用关键帧来实现复杂动画

补间非常适于用来创建简单的过渡效果。更复杂的动画通过使用关键帧，将补间的概念上升了一个层次。关键帧动画包括一系列的值，以及可能各不相同的值与值之间的间隔时间，而非指定一对单值来进行补间。注意*关键帧动画*（key frame animation）这个术语同时用于基于帧的系统和基于时间的系统，这是基于帧的系统的遗留物。

关键帧数据由两部分组成：一个时间（键）列表和一个值列表。值列表表示在相应键的时间点会被使用的属性值；动画系统基于一对键之间的间隔时间值计算补间。

下面的代码片段（来自一个假想的动画引擎）展示了一个关键帧值采样示例，这些数据是用于将一个物体位置抬高并将其移动到远离相机的位置的动画。在一秒的过程中，物体在前 1/4 秒中向上移动，在后 3/4 秒中持续向上移动并向远离相机的位置移动。动画系统在前 1/4 秒中会计算点 (0, 0, 0) 到点 (0, 1, 0) 之间的补间，而在剩余的 3/4 秒中会计算点 (0, 1, 0) 到点 (0, 2, 5) 之间的补间。

```
var keys = [0, 0.25, 1];
var values = [[0, 0, 0],
               [0, 1, 0),
          [0, 2, 5]
              ];
```

关键帧动画支持简单的线性插值算法，也支持基于样条的插值算法这样更复杂的插值算法；换言之，表示关键帧的数据点可以被看作一个线性函数图像上的点，也可以被看作一个更复杂的函数（如三次样条）的图像。尽管补间和关键帧动画都采用了插值，但它们之间存在两个主要的不同点：关键帧动画可以包含两个以上的值，不同关键帧之间的时间间隔可以不同。这使得更强大的效果得以实现，并让动画师可以更好地控制动画。

5.4.1　Keyframe.js——一个简单的帧动画通用库

在学习关键帧动画的示例之前，我们需要首先认识一个用于支持这项技术的动画库。Tween.js 通过用动画队列代替简单的关键值对来支持关键帧动画。尽管如此，我还是认为这样的语法有点麻烦。而且在 Tween.js 中，你也无法改变连续关键点之间的时间间隔。Three.js 实际上提供了内置的、支持关键帧的动画类，但这并不适用于快速手写一些初级的效果，它主要用来支持导入 JSON、COLLADA 以及其他格式的文件。一般来说，关键帧内容本该由 3ds Max、Maya 或 Blender 这类编辑工具直接计算生成，而非手动编写。但是如果有一个可以让程序员方便地将一些简单关键帧连接在一起的途径，也是挺不错的。由于找不到合适的、用于 WebGL 的简易关键帧解决方案，我编写了一个自己的通用库，Keyframe.js。

Keyframe.js 非常简单。它实现了两个类：一个用于控制动画状态（开始、停止、循环逻辑等）的类 `KeyFrameAnimator`，以及一个用于计算两个关键帧之间的补间的类 `Interpolator`。目前这个库只默认支持线性插值。不过 Keyframe.js 允许程序员提供缓动函数，为此我们可以借用 Tween.js 中实现的优秀的 Penner 方程，而无需重复发明轮子。打开文件 Chapter 5/keyframeanimation.html，实际观看 Keyframe.js 的效果。你会看到一个如图 5-4 中的截屏所示的页面。我们可以看到一个远洋冒险的过程：一个箱子随着湍急的水流上下起伏，天空忽明忽暗，预示着暴风雨即将到来。右边的控制面板允许你改变动画时间间隔、开启/关闭不同的动画，以及勾选是否循环。

例 5-5 展示了用于控制箱子动画的代码。首先，我们创建一个 `KF.KeyFrameAnimator` 对象，并用这些参数来初始化它：是否循环、动画时长（以毫秒为单位）、缓动函数（借用了 Tween.js 的），以及赋予 `interps` 参数的一个关键帧插值集合。值得注意的是，与 Tween.js 不同，这里的关键帧和值是列表，而不仅仅是数值对；此外，不同关键帧之间的间隔也是不同的。根据位置插值（`target:group.position`）的详情，箱子在时刻 `t = 0` 到 `t = 0.2` 之间向左前方运行，然后迅速返回起始点（`t = 0.2` 到 `0.25`），在这之后它迅速浸入水中（`t = 0.25` 到 `0.375`）。在 `t = 0.5` 时刻，它回到水面并慢慢下沉（`t = 0.5` 到 `0.9`），最后在 `t = 1.0` 时晃动到原始位置。注意在 Keyframe.js 中，时间点被表示为整个动画时长的一个分量；也就是说，它们的取值范围始终在 0 到 1 之间，因此一个动画帧真正的时间长度等于：

```
time = t × duration
```

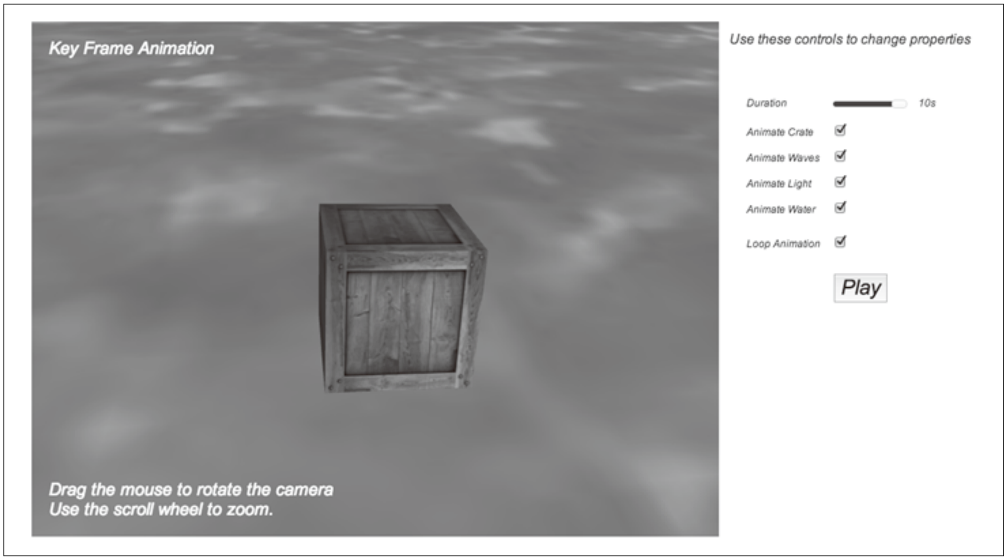


图 5-4：使用关键帧创建复杂动画

还有一个用于旋转的插值序列，用来控制箱子围绕 *x* 轴的倾斜。注意这个差值序列和用于控制位置的序列有不同的关键帧数量；这是有效的，并且可以说是一个特性。位置和旋转的动画故意被设计成存在一些偏差的形式，使得运动效果更加无序。最后我们需要留意缓动函数 `TWEEN.Easing.Bounce.InOut` 的加入，这个缓动函数提供的弹性效果将原本割裂而不协调的平移和旋转效果很好地结合了起来：箱子在水中完美地上下晃动出现。最后，我们只需调用 `start()` 函数来启动整个动画。

例 5-5：箱子的关键帧动画

```
        if (animateCrate)
        {
            crateAnimator = new KF.KeyFrameAnimator;
            crateAnimator.init({
                interps:
                    [
                        {
                            keys:[0, .2, .25, .375, .5, .9, 1],
                            values:[
                                    { x : 0, y:0, z: 0 },
                                    { x : .5, y:0, z: .5 },
                                    { x : 0, y:0, z: 0 },
                                    { x : .5, y:-.25, z: .5 },
                                    { x : 0, y:0, z: 0 },
                                    { x : .5, y:-.25, z: .5 },
                                    { x : 0, y:0, z: 0 },
                                    ],
                            target:group.position
                        },
                        {
                            keys:[0, .25, .5, .75, 1],
```

```
                    values:[
                            { x : 0, z : 0 },
                            { x : Math.PI / 12, z : Math.PI / 12 },
                            { x : 0, z : Math.PI / 12 },
                            { x : -Math.PI / 12, z : -Math.PI / 12 },
                            { x : 0, z : 0 },
                            ],
                    target:group.rotation
                },
            ],
        loop: loopAnimation,
        duration:duration * 1000,
        easing:TWEEN.Easing.Bounce.InOut,
    });
    crateAnimator.start();

}
```

水面和暴风雨的动画也可以用类似的方法来处理，但其他动画都不需要使用缓动函数。一个动画用来使水面上下运动（只需简单地将水面围绕 x 轴旋转）；另一个动画用于创造水面波动，它本质上是通过设置纹理映射的偏移值（`offset` 属性）来“滚动”纹理而实现的；还有一个动画用于通过创建 RGB 值的插值来控制闪光。

这个示例展示了关键帧是如何创建出比 Tween.js 所支持的基本过渡更有趣的效果。关键帧很容易被表示为键值数组，从而允许动画组件顺次设置不同间隔的补间。在实际应用中，程序员很少手动去创建这类动画；事实上，艺术家们会使用专业的工具来完成这些工作。这是开发复杂效果的优先选择，尤其是涉及复杂物体时，这是下一节要讨论的主题。

5.4.2　使用关键帧创建关节动画

到目前为止，我们讨论的动画策略可以用于在场景中使单个对象原地运动（例如旋转）或将其移动到其他位置，但在变换层级的辅助下，它们也可以用来创建复合物体中的复杂运动。

假设我们想要创建一个走动并挥手的机器人。我们需要以层级结构来组织这个机器人的模型：机器人的身体包含上半身和下半身，上半身包括手臂和躯干，手臂包括上臂和下臂，等等。通过正确地构建层级结构并在正确的部位设置动画，我们可以让机器人的手臂和腿动起来。这项通过层级结构来结合离散部件组成主体，并为各级组合体添加动画的技术，被称为*关节动画*（articulated animation）。

Three.js 的示例中提供了一个非常好的关节动画演示。加载 Three.js 的示例文件 examples/webgl_loader_collada_keyframe.html，你会看到一个展示泵内部构造的动画模型。随着泵模型的旋转，它开始装配和拆解自身，暴露出各种部件，如阀门、垫片、齿轮、轴承座、螺栓等，这些部件组成了泵。每个部件都有自身单独的动画；尽管如此，借助 Three.js 的变换层级，每个部件在进行其自身的动画步骤，如打开 / 关闭，以及将一个部件插入另一个部件等时，都可以同时跟随其祖先元素运动。图 5-5 展示了这个泵。

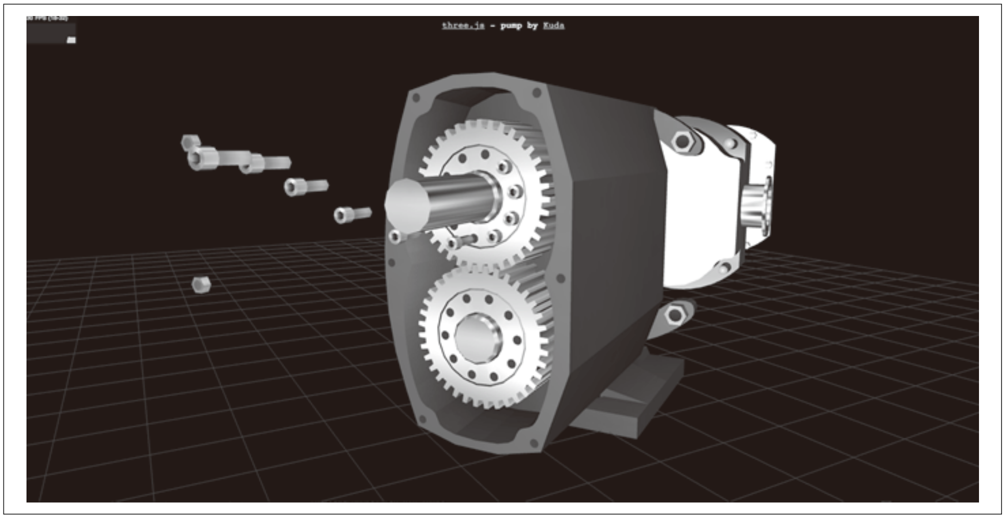


图 5-5：关节动画——使用关键帧和变换层级展示泵的内部工作（使用 Kuda 的开源编辑系统创建的 COLLADA 模型，Kuda 项目主页地址为 https://code.google.com/p/kuda/）

这个泵模型通过 COLLADA 文件格式加载（.dae 文件后缀），这是一个基于 XML 的文本格式，用于描述 3D 内容。COLLADA 可以用来表示单独的模型或整个场景，并支持材质、灯光、相机和动画。我们不打算太深入地了解这些细节，不过有一点需要了解，COLLADA 中的关键帧数据看起来类似下面这个来自泵模型文件（examples/models/collada/pump/pump.dae）的内容片段：

```
<animation id="camTrick_G.translate_camTrick_G">
  <source id="camTrick_G..." name="camTrick_G...">
    <float_array id="camTrick_G..." count="3">0.04166662 ... </float_array>
    <source id="camTrick_G..." name="camTrick_G...">
    <float_array id="camTrick_G..." count="3">8.637086 ... </float_array>
```

COLLADA 中的 <animation> 元素定义了一个动画。两个 <float_array> 子元素分别定义了键和值，这些键值被用于控制一个名为 `camTrick_G` 的物体的 *x* 值变换。键值以秒为单位表示。在整个 7.083 33 秒的过程中，`camTrick_G` 的会沿 *x* 轴从 8.637 086 移动到 0。在这两个关键帧之间还有一个位于 6.5 秒处的关键帧，它对应的 *x* 值为 7.794 443。因此整个动画中，在开始的 6.5 秒，物体的 *x* 变换以非常缓慢的速度进行，随后在剩下的 0.583 33 秒里快速完成整个动画。在整个 COLLADA 文件中，类似的作用于泵模型各个部件物体的动画元素还有很多（总共 74 个）。

例 5-6 展示了示例中用于设置动画的代码片段。这个示例使用了 Three.js 内置的 `THREE.KeyFrameAnimation` 类和 `THREE.AnimationHandler` 类。`THREE.KeyFrameAnimation` 类实现了通用的关键帧动画类型，用于支持 COLLADA 和其他可以包含动画的文件格式。`THREE.AnimationHandler` 是一个单例，用于管理一个动画列表，并在应用的运行循环中保证它们的更新。（这些类的代码可以在 Three.js 项目文件的 src/extras/animation 目录下找到。）

例 5-6：初始化 Three.js 的关键帧动画

```
var animHandler = THREE.AnimationHandler;

for ( var i = 0; i < kfAnimationsLength; ++i ) {

    var animation = animations[ i ];
    animHandler.add( animation );

    var kfAnimation = new THREE.KeyFrameAnimation(
          animation.node, animation.name );
    kfAnimation.timeScale = 1;
    kfAnimations.push( kfAnimation );

}
```

示例在最终调用每个动画的 `play()` 函数来运行动画之前，还需要进行一些设置。`play()` 需要两个参数：一个循环（`loop`）标志和一个可选的开始时间（默认值 0 表示立即开始）：

```
animation.play( false, 0 );
```

这个示例展示了关键帧动画是如何结合变换层级来创建复杂关节效果的。关节动画通常被当作构造机械对象动画的基础。然而在本章的后面部分，关节动画也是用于驱动蒙皮动画下的骨骼的必不可少的因素。

> 和其他随 Three.js 提供的文件格式加载器一样，COLLADA 并不是核心代码包的一部分，而是随示例提供。Three.js COLLADA 加载器的源代码可以在文件 examples/js/loaders/ColladaLoader.js 中找到。COLLADA 格式将会在第 8 章中详细讨论。

5.5 使用曲线和路径创建平滑自然的运动

关键帧非常适合表示一系列具备不同时间间隔的过渡。通过结合关节动画和层级结构，我们可以创建复杂的交互。尽管如此，至今为止我们观看的示例都显得非常机械和人工，这是由于它们都使用了线性插值的缘故。现实世界充满了曲线：汽车沿弯曲的道路行驶，飞机沿着弯曲的航线飞行，子弹沿抛物线掉落，等等。试图用线性插值去模拟这些，只会产生呆板、不自然的结果。我们可以使用一个物理引擎，但对于大多数应用来说，这有点过重了。某些时候我们只需创建一个看起来自然的预定义动画，而无需为模拟物理特效而耗费计算。

关键帧数据不局限于用来描述线性动画。它也可以被看作曲线上的一些点。动画中最常用的曲线是*样条曲线*（spline curve）——一种光滑连续的曲线。被称为 B 样条曲线的某类样条曲线被广泛地应用于计算机图形中，因为它们可以相对快速地被计算出来。我们用一个数据点集合定义 B 样条曲线的基本形状，并在此基础上添加用于调节曲线形状的*控制点*（control point）。图 5-6 中用黑色点标明了简单 B 样条曲线的控制点。如果你使用过像 Adobe Illustrator 这类的专业绘图程序，那肯定对用于修改样条形状的控制点非常熟悉。

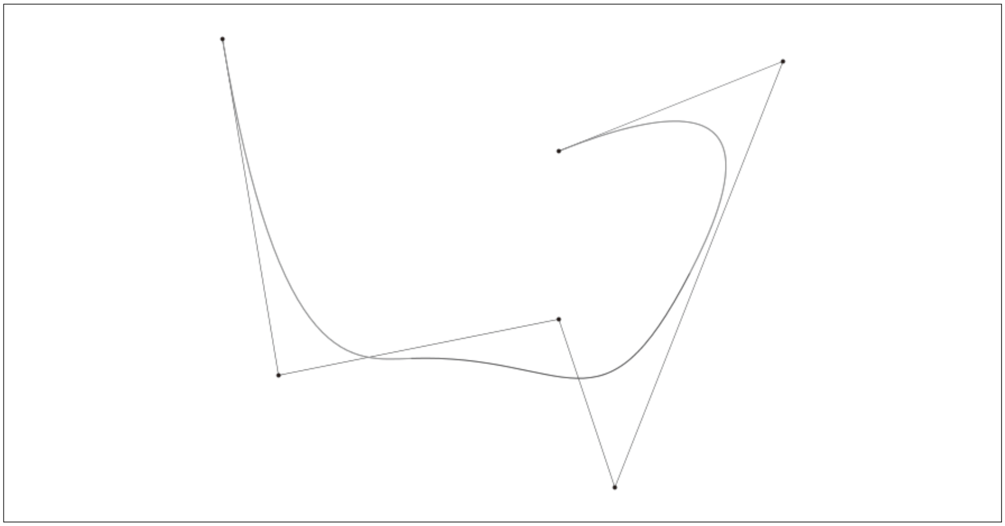


图 5-6：一个 B 样条曲线，由 Wojciech mula 提供（遵循知识共享 CC0 1.0 通用公共领域贡献协议使用）（另见彩插图 5-6）

比起简单的线性插值，样条插值更为复杂，样条插值多项式类似于 Tween.js 中的那些缓动函数实现，并使用位于关键值两侧的数值点来计算出平滑的曲线。在本书中我们不对样条曲线做非常详细的解释。图 5-7 直观地展示了样条曲线是如何工作的：为了计算出曲线上点 `P1` 和点 `P2` 之间的插值，我们同时需要利用控制点 `P0` 和 `P3` 来计算出一个样条曲线上的值。

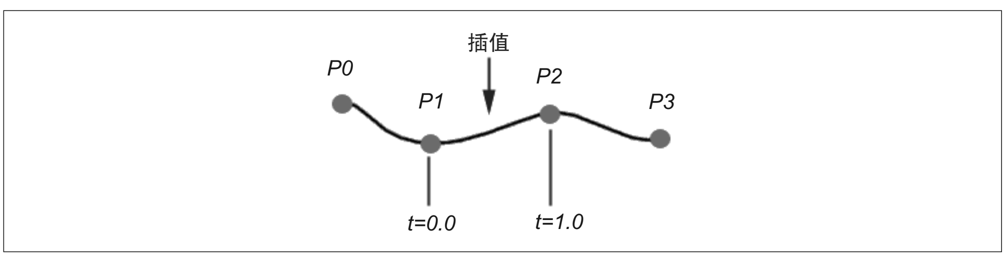
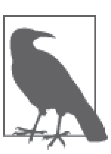


图 5-7：样条插值；经许可进行了修改



样条曲线有很多种，包括 B 样条、三次贝塞尔样条以及 Catmull-Rom 样条（以动画天才、Pixar 的创始人 Ed Catmull 的名字命名）。Catmull-Rom 因其比贝塞尔曲线更加容易构建和计算的特点而流行起来。Three.js 提供了一个内置的、使用了 Catmull-Rom 插值算法的动画类。具体可以查看 Three.js 源文件 src/extras/animations/animation.js。

还有许多优秀的在线 Catmull-Rom 教程，包括 http://flashcove.net/795/cubic-spline-generation-in-as3-catmull-rom-curves/ 和 http://www.mvps.org/directx/articles/catmull/。

样条动画通常需要同时计算方向和位置。例如，如果你需要使某个物体沿一条曲线生成动画，那么为了使得整个运动过程看起来更自然，你需要同时控制它的转弯、倾斜和滚动。这就需要在每个点都重新计算物体的方向。图 5-8 描绘了这个过程。对于曲线的每个点，我们都需要计算其切线（tangent）、法向量（normal）及副法线（binormal）。通俗地说，切线是标示了曲线方向，并和曲线只在一个点上相交的直线。法向量指的是垂直于曲线方向（及切线）的线条。副法线是另外两条线的叉积。这三个向量构成了被称为 TNB 帧的参考帧，它定义了一个物体随路径运动的方向。

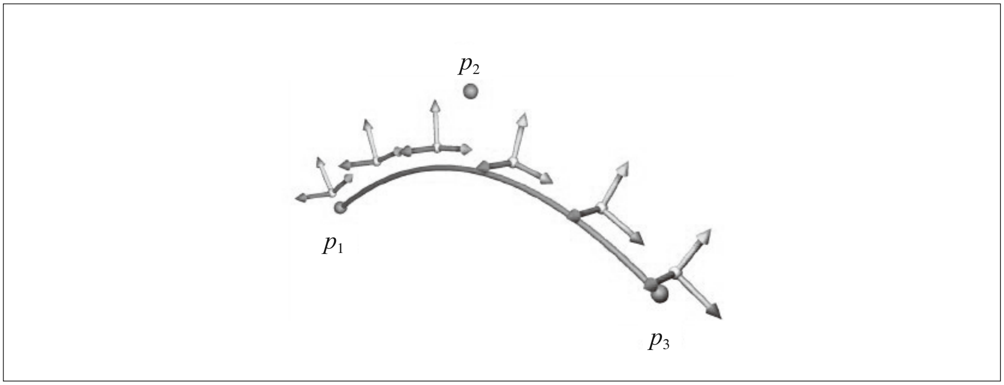


图 5-8：样条动画的坐标帧；切线、法向量和副法线分别用蓝色（向前）、绿色（向上）和红色（向右）箭头来表示；图片由 Cedric Bazillou 提供，经许可进行了修改（另见彩插图 5-8）

在 Three.js 提供的示例中有一个非常好的随路径运动的例子。打开文件 examples/webgl_geometry_extrude_splines.html，确保按钮 Camera Spline Animation View 处于开启（ON）状态，就可以看到如图 5-9 所描绘的动画。相机沿着近处的样条曲线持续改变它的位置和方向。这个示例是程序驱动的，通过代码来计算实时样条插值和 TNB 帧。但它完全可以用一个可复用的路径跟随（path-following）动画类来打包。

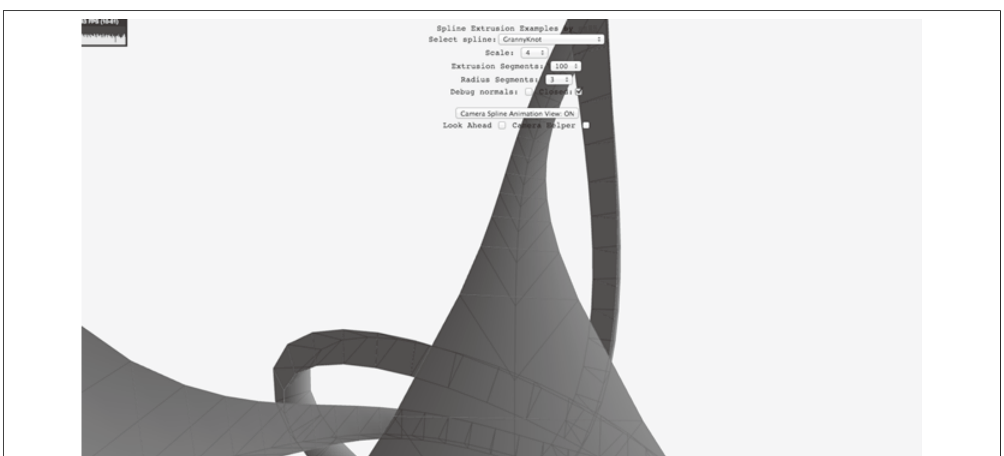


图 5-9：沿路径做动画的相机

5.6 使用变形目标来创建人物和面部动画

关键帧和关节动画非常适用于在场景中移动物体，但许多动画还需要改变物体本身的几何形状。用于实现这类效果的最常用方法是*变形目标动画*（morph target animation），简称*变形*。变形使用基于顶点的插值来改变网格的顶点。通常，网格顶点的一个子集及其索引一起，被当作即将用在补间中的*变形目标*（morph target）存储起来。变形目标各顶点值之间的插值，以及使用了这些插值的动画，组成了网格顶点的变形。

变形目标尤其适合用来表现面部，以及其他不容易用蒙皮动画（在下一节中会说明）来表示的丰富细节。它们非常紧凑，并且不需要由许多面部骨头组成的高度精细的骨骼。此外，它允许动画制作人员从顶点级别以调整网格的方式来创建各种表情。图 5-10 描绘了如何使用变形来创造面部表情。每个表情，如撅嘴和微笑，都是由一个包含了口部和周围区域的顶点集合来表现的。

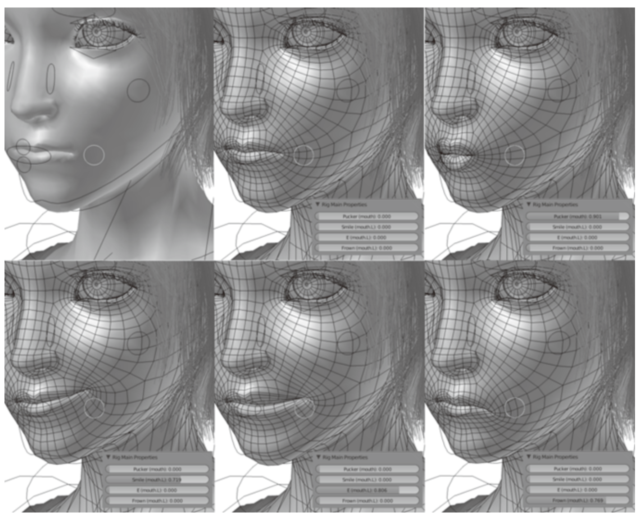


图 5-10：面部变形；遵循知识共享署名－相同方式共享 3.0 未本地化版本协议使用

变形不仅仅可以用于表现面部。Three.js 工程目录中的一些示例使用变形来构建整个角色的动画。图 5-11 描绘了用变形来进行动画的角色。这些角色模型由 id Software 的 MD2 格式的源文件提供。MD2 格式是一种流行的基于变形的格式，它被用于 id Software 公司的游戏（如 *Quake II*）中的玩家角色动画。MD2 文件在这里被转换成 Three.js JSON 文件格式（在第 8 章说明）。



图 5-11：用变形目标创建的角色动画；模型由 MD2 格式转换成 Three.js JSON 格式（“Ogro”角色由 Magarnigal 提供）

加载文件 examples/webgl_morphtargets_md2_control.htm 来观看这些动画。你会看到许多笨拙地转动并往肩膀左右张望的怪物（ogre）角色。键盘上的 WSAD 键可以控制角色在场景中跑动，并从空闲动画切换到走动和转弯的动画队列。效果非常真实。

打开位于 examples/models/animated/ogro/ogro-light.js 的 MD2 文件，感受一下变形对象数据是什么样的。在第 18 行左右，你会看到如下开头的 JSON 属性：

```
"morphTargets": [
{ "name": "stand001", "vertices": [0.6,-2.7,1.5,-5.5,-3.3,-0.6 ...
```

完整的数据占据了很多行。变形目标（`morphTargets`）数组中的每个元素都是一个单独的变形目标；每个变形目标都包含怪物网格的全部顶点，但顶点的位置各不相同。Three.js 通过遍历模型目标集合，使用顶点插值来协调一个目标到下一个的变化。在 `THREE.MD2CharacterComplex` 类（Three.js 示例文件 examples/js/MD2CharacterComplex.js）中，你可以找到为 MD2 角色加载、设置以及添加动画的代码。

> 本示例使用一个由 Klas（昵称 OutsideOfSociety，https://twitter.com/oosmoxiecode，瑞士交互开发团队 North Kingdom 的一名成员）编写的优秀在线工具，将 MD2 格式的文件转换为 Three.js 格式的文件。想了解更多关于这个转换器的信息，请访问 Klas 的博客文章（http://oos.moxiecode.com/blog/index.php/2012/01/md2-to-json-converter）。

5.7 使用蒙皮来构建角色动画

关节动画在表现无生命物体时非常出色，像机器人、汽车、机器，等等。但在表现有机物

体运动的时候则并不理想。微风中摇曳的植物，蹦蹦跳跳的动物，以及跳舞的人们，所有这些都涉及网格的几何形状变化：肢体的扭动，皮肤的延展，肌肉的收缩。你几乎不可能用关节动画的模式来很好地模拟这些事物。因此我们转向另一项被称为*蒙皮动画*（skinned animation，skinning）的技术，这项技术又被称为*骨骼动画*（skeletal animation），或*单网格动画*（single mesh animation）。

蒙皮动画涉及对网格（或者说，蒙皮）上的当前顶点进行实时变换。动画由一个被称为*骨骼*（或支架）的关节物体层级结构来驱动。骨骼仅仅被用作动画的皮下机制，我们不会在屏幕上看到它。骨骼的变化，以及描述骨骼变化是如何影响蒙皮网格上不同区域的额外数据，共同驱动蒙皮动画。图 5-12 描绘了一个简单的骨骼及其相关的蒙皮。

毫不奇怪，骨骼由骨头组成。一个层级结构以你能够直观感受的方式将骨头组织起来。正如那首老歌里所唱的那样：脚骨连接到腿骨，腿骨连接到膝盖骨，等等。和关节动画类似，一个骨头的变换也会影响其子层级。然而和关节动画不一样的是，骨骼是不可见的。

骨骼中的每根骨头都和网格上的一个顶点集合及每个相关顶点的*融合权重*（blend weight，又称*顶点权重*，vertex weight）相关联。融合权重定义了每根骨头与相关顶点的关联程度。每个顶点都可以关联多根骨头，因此每个顶点最终的位置和方向由所有关联骨头的变换，经各自权重处理后的和来决定。这听起来的确很复杂。蒙皮动画多数时候都是使用编辑工具而不是手动创建的。它们涉及的算法同样非常复杂；当前，绝大多数的实时动画引擎在可能的情况下都会利用 GPU 来计算蒙皮动画，包括 Three.js。

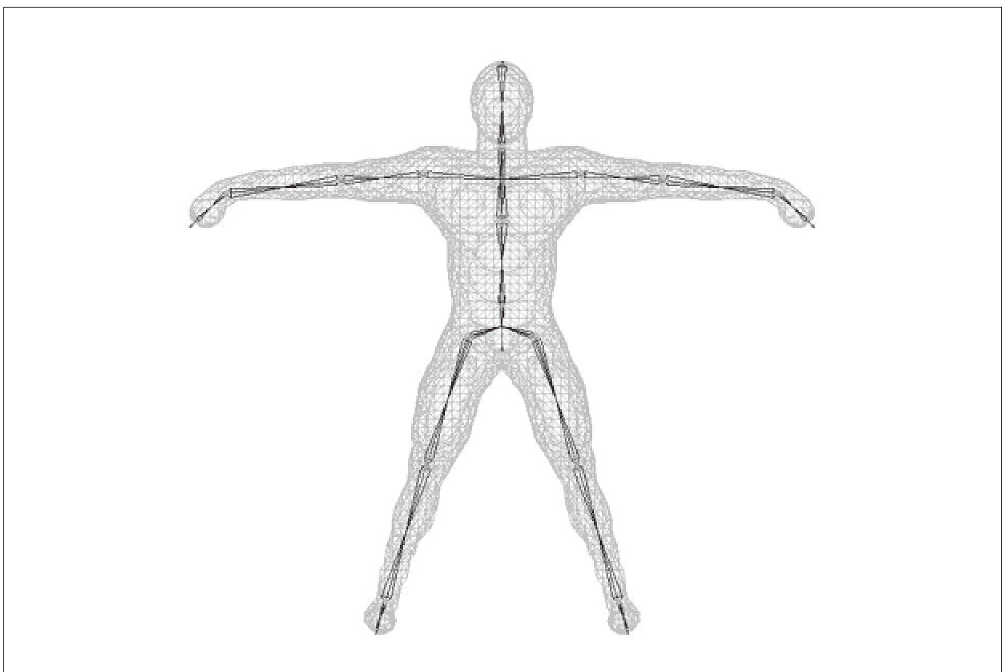


图 5-12：一个包含皮下骨骼的角色网格，适合用蒙皮动画来表现——来自 Frank A. Rivera 创建的一个蒙皮教程

打开 Three.js 工程目录下的文件 examples/webgl_animation_skinning.html，实际观看蒙皮动画的示例。你会在野牛模型上观察到许多真实的效果。点击开始动画，野牛会以看起来非常自然的动作在原地跑动，如图 5-13 所示。

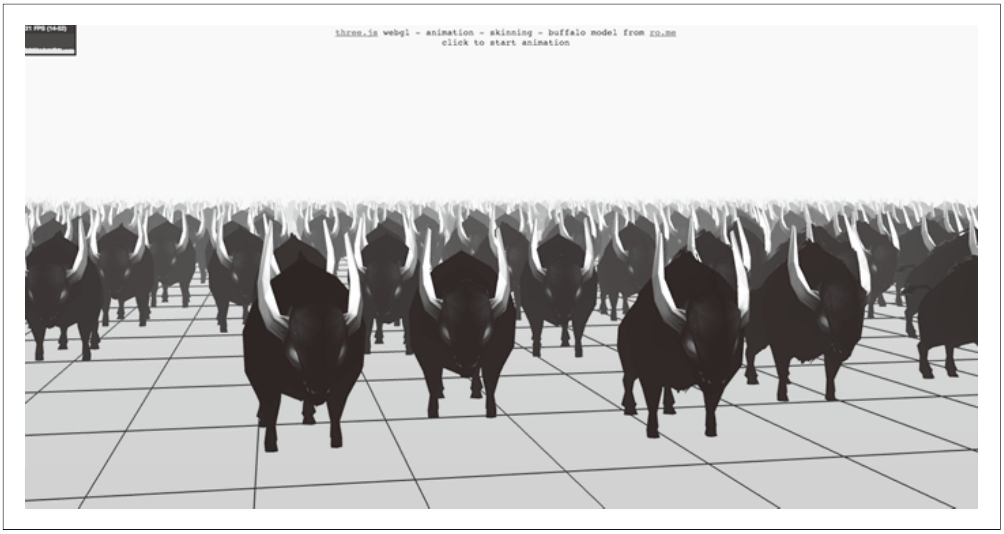


图 5-13：在 Three.js 中使用了蒙皮的网格动画；野牛模型来自 RO.ME

让我们来读一读这个示例的部分代码，以了解 Three.js 是如何实现蒙皮的。首先，我们通过创建一个 `THREE.JSONLoader` 对象，并调用其 `load()` 方法来加载野牛模型。这个类从 Three.js JSON 格式的文件中将模型加载进来。这个格式包含了蒙皮信息以及几何形状信息。

```
var loader = new THREE.JSONLoader();
loader.load( "obj/buffalo/buffalo.js", createScene );
```

`load()` 的第二个参数是一个回调函数，它会在文件被下载并解析完成后调用。例 5-7 展示了回调函数 `createScene()` 的一部分代码，相关的代码行以粗体标出。

例 5-7：用于在文件加载完成后设置蒙皮动画的回调函数

```
function createScene( geometry, materials ) {

    buffalos = [];
    animations = [];

    var x, y,
        buffalo, animation,
        gridx = 25, gridz = 15,
        sepx  = 150, sepz = 300;

    var material = new THREE.MeshFaceMaterial( materials );

    var originalMaterial = materials[ 0 ];
```

```
originalMaterial.skinning = true;
originalMaterial.transparent = true;
originalMaterial.alphaTest = 0.75;

THREE.AnimationHandler.add( geometry.animation );

for( x = 0; x < gridx; x ++ ) {

    for( z = 0; z < gridz; z ++ ) {

        buffalo = new THREE.SkinnedMesh( geometry,
                material, false );

        buffalo.position.x = - ( gridx - 1 ) * sepx * 0.5 +
            x * sepx + Math.random() * 0.5 * sepx;

        buffalo.position.z = - ( gridz - 1 ) * sepz * 0.5 +
            z * sepz + Math.random() * 0.5 * sepz - 500;

        buffalo.position.y =
            buffalo.geometry.boundingSphere.radius * 0.5;
        buffalo.rotation.y = 0.2 - Math.random() * 0.4;

        scene.add( buffalo );

        buffalos.push( buffalo );

        animation = new THREE.Animation( buffalo, "take_001" );
        animations.push( animation );

        offset.push( Math.random() );

    }
```

createScene() 运行一个循环，通过加载进来的一个几何形状来创建野牛网格的多个实例。注意这里创建的网格类型：在这里我们并没有创建在之前的示例中所熟悉的 THREE.Mesh，而是使用了一种不同的网格类型——THREE.SkinnedMesh。这个类型在 Three.js 中会通过一个用于在 GPU 上执行蒙皮动画的特殊顶点着色器来渲染，以达到提升性能的目的。

createScene() 还使用了 Three.js 内置的动画类 THREE.Animation 和 THREE.AnimationHandler。THREE.Animation 实现了通用的关键帧动画，在蒙皮动画中，它被用于驱动骨骼动画。THREE.AnimationHandler 是一个存储了场景中所有动画，并保证它们在应用运行循环中不断更新的单例。我们的回调函数首先通过调用 THREE.AnimationHandler.add()，向其传入几何形状的动画数据，来将动画数据加入动画处理程序队列。随后，这段代码为每个野牛实例创建了一个 THREE.Animation 对象，将使用变量 buffalo 存储的野牛实例和来自 JSON 文件名为 take_001 的动画关联起来。

在动画初始化完成之后，我们开始播放它们。这个应用通过在鼠标点击时调用函数 startAnimation() 来播放动画。如例 5-8 中，startAnimation() 循环访问动画数组，并调用数组中每个元素的 play() 方法。它还为每个动画都赋予了一个各不相同的随机时间偏移，用以避免这些动物的动作完全同步。

例 5-8：播放蒙皮动画

```
function startAnimation() {

   for( var i = 0; i < animations.length; i ++ ) {

       animations[ i ].offset = 0.05 * Math.random();
       animations[ i ].play();
   }

   dz = dstep;
   playback = true;

}
```

如果你对这里的 JSON 动画格式的细节感兴趣，那么可以查看文件 examples/obj/buffalo.js。在整个文件中查找属性 `bones`、`skinWeights` 和 `skinIndices` 来看看骨骼数据是如何表示的；查找属性 `animation`，它包含了用于控制骨骼动画的关键帧层级结构。底层实现的东西非常多，并且 Three.js 在此基础上添加了许多值，而并不仅仅是一个使用 GPU 计算的、基于着色器的蒙皮实现。

5.8 使用着色器来进行动画

到现在为止，我们在本章中所研究过的技术，例如关键帧、补间和蒙皮，都是可以用 JavaScript 来实现的，不过你也可以使用 GLSL 可编程着色器来实现硬件加速的版本。Three.js 中的动画同时使用两种策略：纯 JavaScript 的关键帧系统，以及作为 Three.js 内置材质（如 Phong 和 Lambert）的着色器代码的一部分而实现的变形和蒙皮。如果网格中包含蒙皮和变形的数据（使用前面所描述的 `THREE.SkinnedMesh` 或 `THREE.MorphAnimMesh`），那么 Three.js 会使用这些信息来计算新的顶点位置。

> 如果你对 Three.js 中蒙皮和变形的详细代码有兴趣，打开 Three.js 的源文件 src/renderers/WebGL Shaders.js 并在其中搜索“skin”和“morph”，不过我建议你同时深入了解 GLSL 语言本身以及 Three.js 中的着色器实现。如果你能够对这两方面都有深入了解，这将是非常有意义和有价值的。

除了可以用 GPU 来实现蒙皮之类通用技术的性能优化，我们还可以编写 GLSL 代码来实现任意的效果。也许我们想要创建一个波光闪动的海洋表面，来模拟波浪波动时光的反射和折射；又或许我们想要创建随风摆动的草地。我们可以用纯 JavaScript 代码来实现这些效果，但是 GLSL 更适合用来操作大规模的顶点和图像数据参与的计算。Three.js 项目文件中提供了一个优秀的基于着色器的动画示例。打开示例文件 examples/webgl_shader_lava.html，你会看到一个缓缓旋转的、表面是流动熔岩的圆环形，如图 5-14 所示。

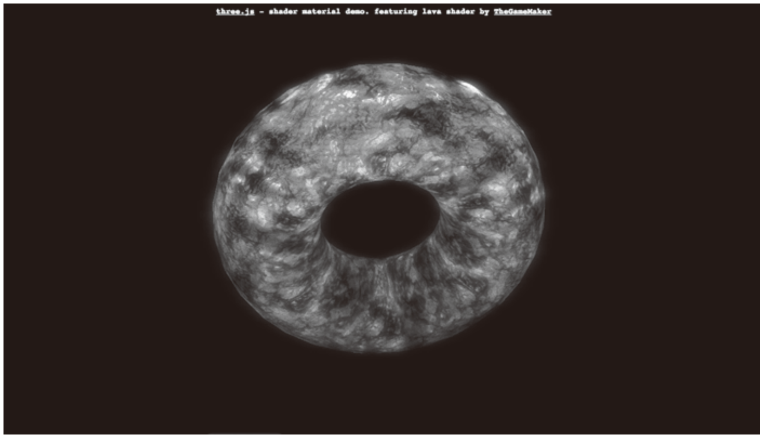


图 5-14：使用 GLSL 创建的熔岩动画效果；着色器代码由 TheGameMaker 提供（http://irrlicht.sourceforge.net/forum//viewtopic.php?t=21057）

流动的熔岩动画代码通过一个包含自定义 GLSL 代码的 THREE.ShaderMaterial 变量来实现。让我们来看看。例 5-9 展示了设置 ShaderMaterial 的代码。许多 uniform 变量被传入着色器。与实现动画相关的重要变量包括时间（time）和两个纹理映射，texture1 和 texture2。稍后你会看到，仅用这三个参数，加上一点由数字提供的特殊效果，就创建了以假乱真的流动熔岩。

例 5-9：创建圆环网格和 ShaderMaterial

```
    uniforms = {

        fogDensity: { type: "f", value: 0.45 },
        fogColor: { type: "v3",
          value: new THREE.Vector3( 0, 0, 0 ) },
        time: { type: "f", value: 1.0 },
        resolution: { type: "v2",
          value: new THREE.Vector2() },
        uvScale: { type: "v2",
          value: new THREE.Vector2( 3.0, 1.0 ) },
        texture1: { type: "t",
          value: THREE.ImageUtils.loadTexture(
            "textures/lava/cloud.png" ) },
        texture2: { type: "t",
          value: THREE.ImageUtils.loadTexture(
            "textures/lava/lavatile.jpg" ) }

    };

  uniforms.texture1.value.wrapS =
    uniforms.texture1.value.wrapT = THREE.RepeatWrapping;
  uniforms.texture2.value.wrapS =
    uniforms.texture2.value.wrapT = THREE.RepeatWrapping;
```

现在所有的 uniform 配置已经就绪，我们可以着手创建着色器材质了。我们需要为构造函数提供顶点和片段着色器代码。注意下面用于完成这件事的技术：我们使用 HTML 中的 `<script>` 元素来存放 GLSL 源代码，并访问 `script` 的 `textContent` 属性来获取 GLSL 文本。对比一下我们之前看到的着色器示例。使用这个方法，我们可以以更直截了当的方式来编写着色器代码，而无需使用转义换行符来连接多行文本字符串。稍后我们将看到具体的 GLSL 代码。

```
var size = 0.65;

material = new THREE.ShaderMaterial( {

    uniforms: uniforms,
    vertexShader: document.getElementById(
      'vertexShader' ).textContent,
    fragmentShader: document.getElementById(
      'fragmentShader' ).textContent

} );
```

随后我们创建一个具备 `THREE.ShaderMaterial` 材质的圆环网格，并将其添加到场景中。

```
mesh = new THREE.Mesh(
  new THREE.TorusGeometry( size, 0.3, 30, 30 ),
    material );
mesh.rotation.x = 0.3;
scene.add( mesh );
```

着色器算法非常清晰。它使用了两个纹理映射，一个用于提供基础的熔岩颜色和视觉效果，另一个云纹理作为“噪声”的来源，用于对基础纹理添加扰动，以形成流动的视觉效果。两个纹理如图 5-15 所示。

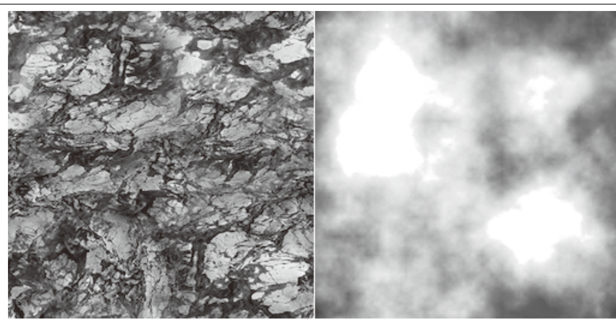


图 5-15：用于熔岩和噪声的纹理映射

顶点着色器的 GLSL 代码非常简单，参见例 5-10。与大多数着色器一样，它使用模型视图矩阵和投影矩阵来乘上顶点，将其映射到屏幕空间，并将这个值输出到一个内置的 GLSL 变量 `gl_Position` 中。在此之前，我们定义了一个 varying 参数——`vUv`。这是每个顶点的纹理坐标，顶点着色器向片段着色器输出这个值（我们稍后将会看到），供片段着色器后续计算使用。这个着色器还允许输入一个缩放参数，用来控制纹理坐标的缩放。

正如之前提到的，GLSL 源代码嵌入在一个 <script> 元素中，因此我们可以在没有引号和换行符这类干扰的情况下流畅地阅读这些代码。这里的诀窍在于我们使用了一个不同的脚本类型标签，在这个例子中我们使用了 x-shader/x-vertex。浏览器无法识别这个类型；我们用它来表明这不是一个 JavaScript 语言脚本。

例 5-10：嵌入在 HTML<script> 元素中的顶点着色器代码

```
<script id="vertexShader" type="x-shader/x-vertex">

    uniform vec2 uvScale;
    varying vec2 vUv;

    void main()
    {

        vUv = uvScale * uv;
        vec4 mvPosition = modelViewMatrix * vec4( position, 1.0 );
        gl_Position = projectionMatrix * mvPosition;

    }

</script>
```

片段着色器代码承担了大部分的工作。例 5-11 展示了这段代码。在定义了对应 JavaScript 中参数的 uniform 参数之后，我们定义了一个 varying 参数——vUv，用于匹配来自顶点着色器的输出。

例 5-11：基于着色器的动画中的片段着色器代码

```
<script id="fragmentShader" type="x-shader/x-fragment">

    uniform float time;
    uniform vec2 resolution;

    uniform float fogDensity;
    uniform vec3 fogColor;

    uniform sampler2D texture1;
    uniform sampler2D texture2;

    varying vec2 vUv;
```

下面是片段着色器的主要代码。它的要点在于 texture1，即云纹理，它被用作噪声的来源，来对从 texture2（熔岩纹理）获取颜色值的纹理坐标产生轻微扰动。（GLSL 函数 texture2D() 通过一个给定的 2D 纹理坐标从纹理中获取颜色数据。）通过将噪音纹理坐标和当前数值相乘，并加入一些凭经验确定的偏移（例如 1.5, −1.5），我们可以得到流动的效果。像素点的颜色值最后被保存在内置的 GLSL 变量 gl_FragColor 中。

```
void main( void ) {

            vec2 position = -1.0 + 2.0 * vUv;

            vec4 noise = texture2D( texture1, vUv );
            vec2 T1 = vUv + vec2( 1.5, -1.5 ) * time  *0.02;
```

```
vec2 T2 = vUv + vec2( -0.5, 2.0 ) * time * 0.01;

T1.x += noise.x * 2.0;
T1.y += noise.y * 2.0;
T2.x -= noise.y * 0.2;
T2.y += noise.z * 0.2;

float p = texture2D( texture1, T1 * 2.0 ).a;

vec4 color = texture2D( texture2, T2 * 2.0 );
vec4 temp = color * ( vec4( p, p, p, p ) * 2.0 ) +
    ( color * color - 0.1 );

if( temp.r > 1.0 ){ temp.bg += clamp( temp.r - 2.0, 0.0, 100.0 ); }
if( temp.g > 1.0 ){ temp.rb += temp.g - 1.0; }
if( temp.b > 1.0 ){ temp.rg += temp.b - 1.0; }

gl_FragColor = temp;
```

到这里，流动的熔岩效果就完成了。然而，着色器还增加了一个雾效（fog effect）。保存在 `gl_FragColor` 中的值最终会和一个由向着色器传入的 fog 参数计算出来的雾效值混合，得出像素最终的颜色值，并输出到内置的 GLSL 变量 `gl_FragColor` 中。到这里，整个过程就结束了。

```
        float depth = gl_FragCoord.z / gl_FragCoord.w;
        const float LOG2 = 1.442695;
        float fogFactor = exp2( - fogDensity * fogDensity * depth *
            depth * LOG2 );
        fogFactor = 1.0 - clamp( fogFactor, 0.0, 1.0 );
        gl_FragColor = mix( gl_FragColor,
            vec4( fogColor, gl_FragColor.w ), fogFactor );

    }

</script>
```

最后，在我们的运行循环中通过不断更新各项值来驱动动画。Three.js 让这项工作变得非常简单；它在渲染器每次更新时都自动向 GLSL 着色器传递全部的 uniform 值。我们需要做的仅仅是在 JavaScript 代码中设置这些属性。在这个示例中，`render()` 函数在每个动画帧执行的时候都会被调用。注意用粗体标出的代码行。

```
function render() {

    var delta = 5 * clock.getDelta();

    uniforms.time.value += 0.2 * delta;

    mesh.rotation.y += 0.0125 * delta;
    mesh.rotation.x += 0.05 * delta;

    renderer.clear();
    composer.render( 0.01 );

}
```

不可否认，编写这样一个动画需要比较高的艺术水平。我们不仅仅需要学习详细的 GLSL 语法和内置函数，还需要掌握一些深奥的计算机图形算法。但是如果你有这方面的爱好，它会让你非常有成就感。互联网上有非常多的信息以及有用的示例，可以帮助你入门。

5.9 小结

正如我们所看到的，在 WebGL 中使 3D 内容动起来的方法有很多。从本质上来说，这些动画由新的浏览器函数 `requestAnimationFrame()` 来驱动，这个函数使得用户可以在整个页面中进行实时同步绘制。此外，为了驱动动画，我们有从简单到复杂的多种选择，取决于想要实现的效果。内容可以通过逐帧用程序修改来产生动画，也可以使用一些数据驱动的方式，如补间、关键帧、变形和蒙皮。我们可以通过将关键帧和路径跟踪结合来获得更自然的运动效果。我们还可以使用着色器来在 GPU 中进行内容动画，以实现更多的可能。用于 WebGL 动画的工具和库都在持续更新，并没有一个最终确定的选择。然而这意味着存在许多可能性，也意味着少了许多障碍，而这要感谢 JavaScript 和开源。

第 6 章

CSS3：高级页面效果

前几章展示了如何使用 WebGL 来创建惊人的、硬件加速渲染的 3D 物体、场景和动画。尽管 WebGL 非常强大，但在本书写作时它依然存在一个基本的局限，那就是你不能将任意的 HTML 内容作为一个纹理映射到一个 3D 物体的表面。如果你希望为页面上的元素添加我们在前几章中看到的 3D 效果，需要使用另一项 HTML5 的新技术：CSS3。

有了 CSS3，单个元素或整个页面都可以通过动画、图片滤镜，以及 2D 或 3D 变换变得生动起来。这些特性使得我们可以创建各种各样用于简单游戏、广告宣传和直观用户界面的 3D 效果。WebGL 至少需要基本的 3D 编程知识并掌握一个如 Three.js 这样的库，而相比之下使用 CSS3 则只需了解标签、CSS 和基本的 JavaScript，或许还需要像 jQuery 这类框架的一点辅助。这让 CSS3 的开发比 WebGL 更加简单；然而，开发者只能访问浏览器内置的特性。换句话说，3D CSS 以牺牲强大和灵活性来换取了简单易用。

CSS3 的 3D 特性源于 Apple 最初为其核心动画框架编写的 3D 变换，它支持了如 iOS 天气应用中的屏幕转换（如图 6-1）这类现今流行的用户界面效果。CSS3 中的 3D 方案最初由 WebKit 开发团队在 2009 到 2010 年间提出，并被 Apple 的 Safari 团队首次应用于市场，用在 Mac OS 和 iOS 平台上。这些特性随后被 Chrome 采用，并最终为所有的浏览器厂商采纳。



图 6-1：iOS 天气应用中的屏幕转换

为 HTML 元素添加 3D 效果的能力，为 Web 页面内容开启了同样的可能性。图 6-2 展示了 Snowstack，一个由 Safari 团队开发的演示（http://www.satine.org/research/webkit/snowleopard/snowstack.html）。Snowstack 是一个使用纯 HTML、3D CSS 和 JavaScript 创建的照片查看视觉效果库，它用于以透视的方式渲染一个 Flickr 消息队列。有了 Snowstack，用户可以使用键盘上的方向键来控制浏览一个看似无尽的照片瓷砖墙。这个应用可以在所有的浏览器和设备中运行。虽然 Snowstack 本身的确是一个非常有技术含量的演示，但它旨在使用 3D CSS 来对海量信息进行可视化和浏览。



图 6-2：Snowstack，一个基于 CSS3 的 3D 照片查看器

许多开发者为创建创新的 Web 内容而研究 3D CSS。除了简单地为平面瓷砖添加变换之外，一些程序员想出了如何用这项技术来模拟完整的 3D 物体，我们在本章后面将会看到，

有人甚至大胆地使用 3D CSS 来构建了一个第一人称射击游戏的原型！3D CSS 还能与 WebGL 结合使用，后者用于处理真实的 3D 渲染任务，而前者用于覆盖或整合用户界面的 HTML 元素。

CSS3 是允许往页面元素上添加动态效果的一系列标准。本章涵盖了多种用于创建 3D 效果的 CSS 技术。

- *CSS 变换*

  作用于整个元素的 3D 操作（平移、旋转、缩放）。

- *CSS 过渡*

  随时间变化作用于 CSS 属性的简单变化。如补间（在上一章讨论过）一样，CSS 过渡非常适合用于一次性的效果。

- *CSS 动画*

  随时间作用于 CSS 属性的复杂效果，使用关键帧数据。

6.1 CSS变换

3D CSS 开发的核心是使用 CSS 变换来操作页面元素的能力。CSS 变换规范（http://www.w3.org/TR/css3-transforms/）代表了之前使用了 CSS 的 3D 和 2D 工作在改变页面元素位置、旋转、缩放以及其他外观属性上的趋同。它们使用变换操作来封盖这些属性，而非简单的 left/top 和 width/height 属性。

让我们来复习一下，3D 图形使用一个三维的坐标系统，它采用了第三个坐标轴 *z* 来表示距离屏幕位置的远近，据此创建了深度的观感。图 6-3 描绘了用于 CSS 的 3D 坐标系统。注意，与常规的 3D 系统不同，*y* 轴的正方向向下而不是向上。这是为了和 Web 浏览器页面和窗口坐标系中使用的 2D *xy* 系统保持一致。

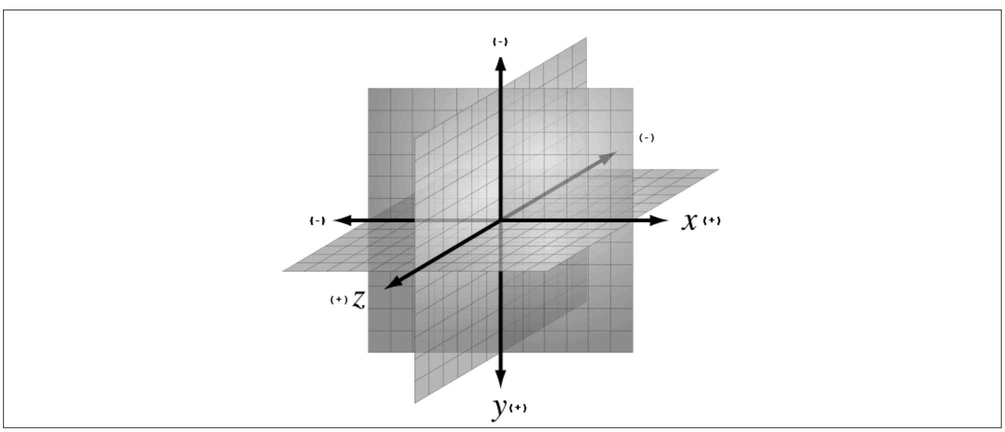


图 6-3：CSS 的 3D 坐标系统，*y* 轴正方向指向下（改编自 http://bit.ly/wikimedia-3d-coordinate；遵循知识共享署名 – 相同方式共享 3.0 未本地化版本协议使用）

6.1.1 使用3D变换

你可以像指定其他 CSS 一样使用属性来指定 CSS 3D 变换。CSS3 标准中定义了许多用于变换元素的属性。我们先来看一个示例。图 6-4 描绘了被赋予不同变换的三个元素：平移、旋转和缩放。

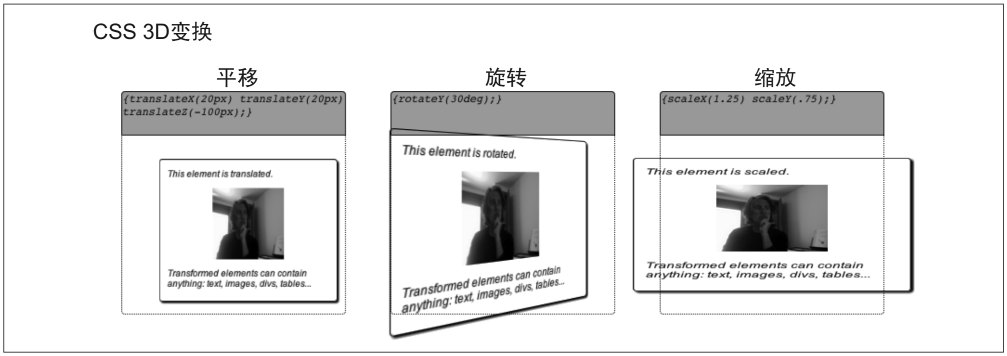


图 6-4：CSS 3D 变换——平移、旋转和缩放

这个示例的源代码可以在文件 Chapter 6/css3dtransforms.html 及其引用的 CSS 文件 css/css3dtransforms.css 中找到。例 6-1 展示了定义第一个 DIV 元素的代码片段，它被赋予了一个 3D 变换。

例 6-1：带 CSS 3D 变换的元素

```
<div id="card1" class="container perspective">
    <div class="legend">
    Translate
    </div>
    <div class="code">{translateX(20px) translateY(20px) translateZ(-100px);}</div>
    <div class="cardBorder">
        <div class="card translate">
          <p>This element is translated.</p>
          <img width=96 height=96 src="../images/HTML5rawkes.png"></img>
          <p>Transformed elements can contain anything: text, images,
             divs, tables...</p>
        </div>
    </div>
</div>
```

粗体字表明了作用于中间的 DIV 元素的两个类：`card` 和 `translate`。`card` 类定义了页面上三个“卡片”的通用属性，例如实线边框、投影以及圆角。`translate` 类定义了 3D 变换。例 6-2 展示了这两个类的 CSS 定义，以及 `cardBorder`，后者用于定义卡片父元素的点线变量，以表明没有添加变换属性的元素的原始位置。现在我们可以暂时忽略这个声明中的 `-moz-transform-style` 属性。它们的作用是保证这些特效在 Firefox 浏览器中也可以正常运行，我会在下一节关于透视的讲述中详细解释这个属性。

例 6-2：定义了一个平移变换的 CSS

```
.cardBorder {
```

```
    position: absolute;
    width: 100%;
    height: 80%;
    top:30%;
    border:1px dotted;
    border-radius:0 0 4px 4px;
    -moz-transform-style: preserve-3d;
}

.card {
    position: absolute;
    width: 99%;
    height: 99%;
    border:1px solid;
    border-radius: 4px;
    box-shadow: 2px 2px 2px;
    -moz-transform-style: preserve-3d;
}

.translate {
    -webkit-transform: translateX(20px) translateY(20px) translateZ(-100px);
       -moz-transform: translateX(20px) translateY(20px) translateZ(-100px);
         -o-transform: translateX(20px) translateY(20px) translateZ(-100px);
            transform: translateX(20px) translateY(20px) translateZ(-100px);
}
```

`translate` 类通过设置它的 `transform` 属性来指定一个 CSS 3D 变换。在这个示例中，元素在 *x* 和 *y* 方向上分别平移了 20 像素，并沿 *z* 轴负方向移动了 100 像素（向屏幕内侧）。总的来说，你可以使用 `transform`，通过将以下一个或多个*变换方法*（transform method）作用于元素来创建变换。除了平移，CSS 还支持旋转和缩放，任意的矩阵变换，以及透视投影。CSS 3D 变换方法如表 6-1 所示。

表6-1：CSS 3D变换方法

方　　法	描　　述
`translateX(x)`	沿 *x* 轴平移
`translateY(y)`	沿 *y* 轴平移
`translateZ(z)`	沿 *z* 轴平移
`translate3d(x,y,z)`	沿 *x*、*y*、*z* 三轴平移
`rotateX(angle)`	绕 *x* 轴旋转
`rotateY(angle)`	绕 *y* 轴旋转
`rotateZ(angle)`	绕 *z* 轴旋转
`rotate3d(x,y,z,angle)`	绕 *x*、*y*、*z* 三轴旋转
`scaleX(x)`	沿 *x* 轴缩放
`scaleY(y)`	沿 *y* 轴缩放
`scaleZ(z)`	沿 *z* 轴缩放
`scale3d(x,y,z)`	沿 *x*、*y*、*z* 三轴缩放
`matirx3d(...)`	用 16 个数值定义任意的 4×4 变换矩阵
`perspective(depth)`	定义透视投影的深度像素值

第二个和第三个卡片以类似的方式进行变换，通过使用 CSS 中定义的 rotate 和 scale 类：

```
.rotate {
    -webkit-transform: rotateY(30deg);
       -moz-transform: rotateY(30deg);
         -o-transform: rotateY(30deg);
            transform: rotateY(30deg);
}

.scale {
    -webkit-transform: scaleX(1.25) scaleY(.75);
       -moz-transform: scaleX(1.25) scaleY(.75);
         -o-transform: scaleX(1.25) scaleY(.75);
            transform: scaleX(1.25) scaleY(.75);
}
```

旋转值可以被指定为角度、弧度或百分度（gradian，圆周的 1/400），例如 90deg、1.57rad 或 100grad。缩放值是用于乘在每个轴上的标量（即一个没有进行过缩放的元素在每个轴上的缩放值都是 1）。

> 注意浏览器私有前缀在 CSS 中的使用（例如 -webkit-transform）。这对跨浏览器支持是必要的，因为 CSS 变换在好几年的时间里都被作为实验性的特性来支持。这真麻烦，但许多 CSS 特性都需要通过这样添加前缀的方式来使用，而开发者们也已经习惯了处理它们。如果你对这些重复的工作感到厌烦，那么可能需要一个可以自动生成样式表的工具，例如 LESS（http://lesscss.org/），来使得这项工作不那么痛苦。偶尔为了简洁，在我们的示例中我会省略掉这些浏览器私有前缀，但请务必确保在你的代码中使用它们。

CSS 支持一个额外的属性，transform-origin，这个属性允许开发者指定变换的原点。这个属性的默认值是 50% 50% 0，即坐标系的中心点。通过改变这个属性，你可以让物体围绕非中心点旋转。transform-origin 可以使用任意的 CSS 偏移单位来指定，如 left、center、right、%，或者一个 CSS 距离值（像素、英寸、em 空间，等等）。

6.1.2　添加透视

你应该已经注意到，在上面的示例中类 perspective 作用于每个顶层 DIV 元素。无论使不使用透视投影，你都可以应用 CSS 3D 变换，尽管使用透视投影会使得效果看起来更令人满意。

在 CSS3 中定义透视投影非常简单。例 6-3 展示了用于定义透视的 CSS。

例 6-3：CSS 透视属性

```
.perspective {
    -webkit-perspective: 400px;
       -moz-perspective: 400px;
         -o-perspective: 400px;
            perspective: 400px;
}
```

```
.noperspective {
    -webkit-perspective: 0px;
       -moz-perspective: 0px;
         -o-perspective: 0px;
            perspective: 0px;
}
```

我们定义了一个 CSS 类——`perspective`，来用于我们希望应用透视投影的元素。我们提供的值代表从视口平面到 *xy* 平面（`z=0`）的距离。透视可以使用任意的 CSS 距离单位来定义：像素、点、英寸、em 空间，等等。这个 CSS 文件还定义了另一个类——`noperspective`，用于清除元素上原先可能带有的透视。这个类中的透视值被设为默认值 0。

尽管 CSS 透视的细节和 WebGL 中的透视并不相同，但它们的概念是一致的。如果你需要重温一下这方面的知识，请翻回到第 1 章，那里有详细的讨论。

为了更好地展现添加了透视和没有添加透视的元素的区别，让我们来看一个例子。打开示例文件 Chapter 6/css3dperspective.html。你会看到两个卡片。左边的卡片添加了透视，而右边的没有。两者之间的唯一区别在于是否使用了 CSS `perspective` 属性；两个卡片都围绕 *y* 轴旋转了 30 度。然而，在没有使用透视的情况下，右边的元素看起来仅仅是在水平方向上被压扁了，而不是旋转。如图 6-5 所示。

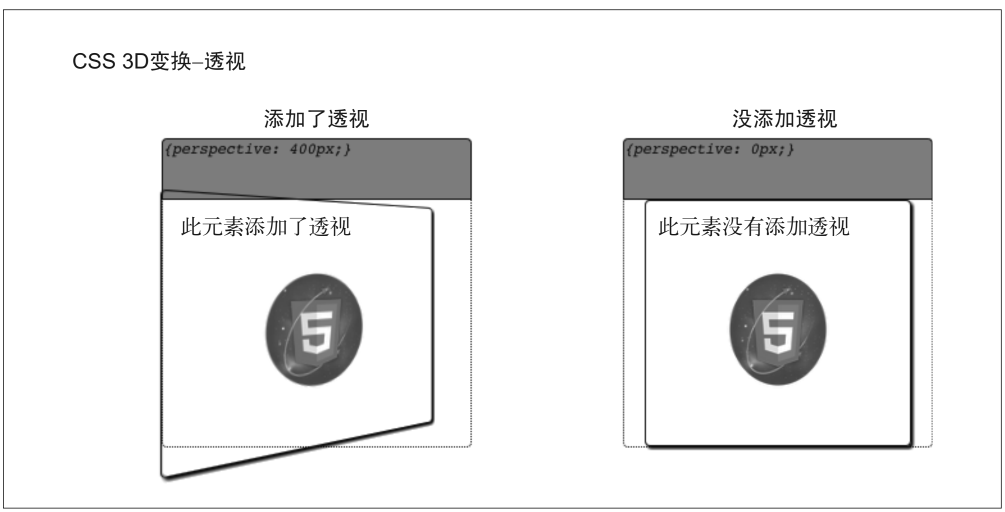


图 6-5：CSS 变换和透视——左边的元素有透视，右边的元素则没有（HTML5 Rawkes Logo 由 Phil Banks 提供）

你还可以使用表 6-1 中描述的 `perspective()` 变换函数将透视应用于元素。然而，在实际应用中，你最好还是使用两个不同的属性来分别表示透视值和变换值。否则，在每次改变其他变换函数值的时候，你可能都需要重新将透视值应用到元素上。

6.1.3 创建变换层级

CSS 3 允许 3D 变换通过 DOM 对象层级继承。一个带 3D 变换的元素可以选择继承或不继承其祖先元素的变换，具体取决于 `transform-style` 属性的值。

图 6-6 描绘了 `transform-style` 是如何用于创建一个变换层级的。每个卡片（`card`）元素都围绕 *y* 轴旋转了 30 度。每个卡片都有一个独立围绕 *y* 轴旋转了 30 度的子卡片（`childCard`）。注意左边的子卡片距它的父元素平面旋转了 30 度，而右边的子卡片与它的父元素处于同一平面上。

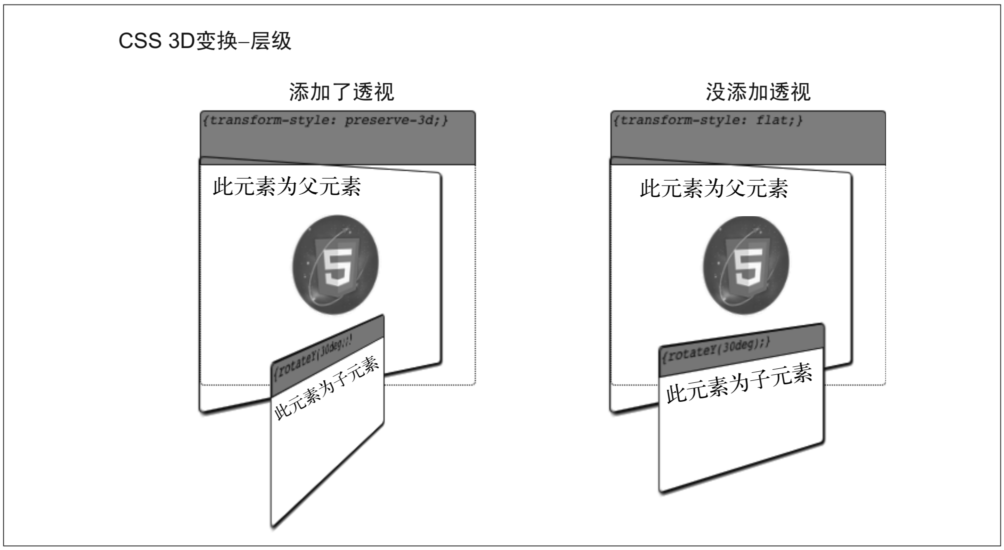


图 6-6：使用 CSS 创建 3D 变换层级

示例代码可以在文件 Chapter 6/css3dhierarchy.html 和 css/css3dhierarchy.css 中找到。HTML 定义了两个基本相同的 DOM 元素层级，唯一的区别在于第一个卡片使用了 `hierarchy` 类，而第二个使用了另一个叫 `nohierarchy` 的类。

```
<div id="hierarchy1" class="container perspective">
    <div class="legend">
    With Hierarchy
    </div>
    <div class="code">{transform-style: preserve-3d;}</div>
    <div class="cardBorder">
        <div class="card hierarchy rotate">
          <p>This element is a parent.</p>
          <img width=96 height=96 src="../images/HTML5rawkes.png"></img>
          <p></p>
            <div class="childCard rotate">
            <div class="code">{rotateY(30deg);}</div>
                <p>This element is a child.</p>
            </div>
        </div>
    </div>
```

```
</div>

<div id="hierarchy2" class="container perspective">
    <div class="legend">
    Without Hierarchy
    </div>
    <div class="code">{transform-style: flat;}</div>
    <div class="cardBorder">
        <div class="card nohierarchy rotate">
        <p>This element is a parent.</p>
        <img width=96 height=96 src="../images/HTML5rawkes.png"></img>
          <p></p>
            <div class="childCard rotate">
              <div class="code">{rotateY(30deg);}</div>
              <p>This element is a child.</p>
            </div>
        </div>
    </div>
</div>
```

类 hierarchy 和 nohierarchy 的 CSS 定义如下：

```
.hierarchy {
     -webkit-transform-style: preserve-3d;
        -moz-transform-style: preserve-3d;
          -o-transform-style: preserve-3d;
             transform-style: preserve-3d;
}

.nohierarchy {
     -webkit-transform-style: flat;
        -moz-transform-style: flat;
          -o-transform-style: flat;
             transform-style: flat;
}
```

transform-style 属性接受两个值：flat（默认），这个值指定了后代 DOM 元素不会应用父元素的变换；preserve-3d，这个值会告诉浏览器将上级元素的变换应用于后代元素。通过在整个应用中使用 preserve-3D，应用可以创建一个 3D 物体的深度层级，尤其在结合了本章所述的另外两项技术的情况下。

浏览器兼容性警告：在本节的第一个示例中，我们一笔带过了 card 和 cardBorder 的 CSS 类定义中的一个细节，它们都包含如下语句：

```
-moz-transform-style: preserve-3d;
```

显然 Firefox 浏览器并不像基于 WebKit 的浏览器那样将 transform-style 属性层层传播下去。你必须为每一个后代元素显式地指定它，否则不仅子元素的变换会失效，透视也被禁用了。解决方案是将 DOM 层级中每个后代的 transform-style 属性都设为 preserve-3d。这虽然很麻烦，但是必要的。

这种情况最坏的部分在于每个浏览器对这个语句的解释都不尽相同。显然 IE10 就完全不支持这个特性，不过 IE11 中会增加对它的支持。

6.1.4 控制背面渲染

在传统的 3D 渲染中，当一个多边形背向观察者时，渲染系统可以选择显示或不显示该多边形的背面（backface），这取决于开发者的设置。CSS3 变换同样提供了这项能力。如果一个元素旋转至背面朝前，它会基于 `backface-visibility` 变换属性来决定是否显示。

CSS3 背面渲染对于构建双面物体非常重要。假设我们想要创建一个类似图 6-1 描绘的 iOS 天气应用中的屏幕翻转过渡效果。那么创建这样一个效果，需要我们小心地组织标签，以及正确地应用 `backface-visibility`。图 6-7 描绘了在实际中如何使用这项技术。

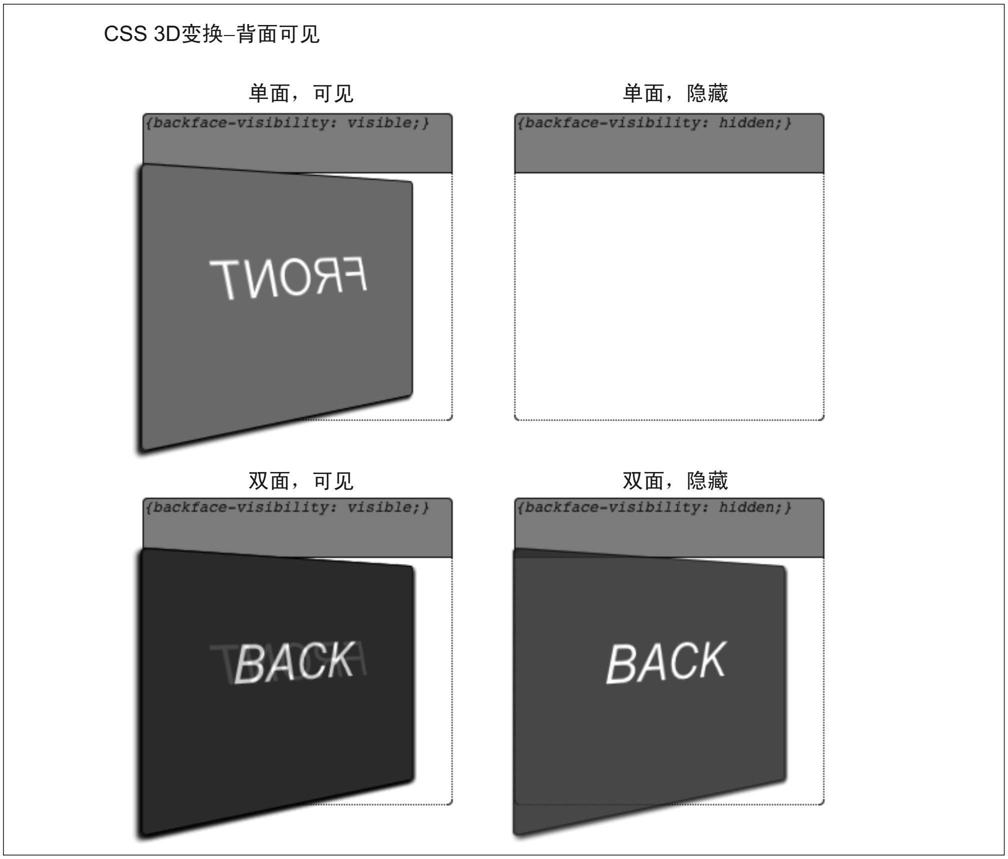


图 6-7：使用背面可见属性来创建双面物体

打开文件 Chapter 6/css3dbackfaces.html 来实际观看背面渲染的效果。页面上有四个卡片。上面一行是两个单面的卡片，分别以背面可见和背面不可见的模式渲染。左上的卡片旋转到背对观察者的角度，并以背面可见的方式渲染；右上的卡片旋转到背对观察者的角度，并以背面不可见的方式渲染。注意，我们可以看到左上的卡片，但文字“FRONT”是反过来的，而右上的卡片不可见。

在下面一行我们可以看到两个双面的卡片，同样分别以背面可见和不可见的方式渲染。

两个物体同样被旋转到背对观察者的角度。然而，这两个卡片都定义了一个包含文字“BACK”的额外元素，用来转向观察者，以模拟双面的物体。左下的卡片开启了背面可见，并设置了一个 0.8 的透明度值，所以我们可以透过表面看到翻转的文字“FRONT”。相反，右下的卡片关闭了背面可见，因此卡片背对观察者的一面（“FRONT”面）被隐藏了。右下的卡片展示了用于模拟双面物体的正确技术，让我们来看看它的代码。

例 6-4 展示了这个页面的 HTML 代码。背面可见的元素通过 `backface` 类来定义，背面隐藏的元素通过 `nobackface` 类来定义。为了创建下面一行的双面卡片，我们实际上需要创建两个卡片元素：分别用于正面和背面。右下的卡片通过将 `nobackface` 类和其他类组合起来，创建了一个无论哪一面朝向观察者都可以正常显示的卡片。

例 6-4：构建一个双面的 HTML 元素

```
<div id="backface1" class="container perspective ">
    <div class="legend">
    One-Sided, Visible
    </div>
    <div class="code">{backface-visibility: visible;}</div>
    <div class="cardBorder">
        <div class="card backface frontside">
        FRONT
        </div>
    </div>
</div>

<div id="backface2" class="container perspective ">
    <div class="legend">
    One-Sided, Hidden
    </div>
    <div class="code">{backface-visibility: hidden;}</div>
    <div class="cardBorder">
        <div class="card nobackface frontside">
        FRONT
        </div>
    </div>
</div>

<div id="backface3" class="container perspective ">
    <div class="legend">
    Two-Sided, Visible
    </div>
    <div class="code">{backface-visibility: visible;}</div>
    <div class="cardBorder">
        <div class="card backface frontside">
        FRONT
        </div>
        <div class="card backface backside">
        BACK
        </div>
    </div>
</div>

<div id="backface4" class="container perspective ">
    <div class="legend">
```

```
    Two-Sided, Hidden
    </div>
    <div class="code">{backface-visibility: hidden;}</div>
    <div class="cardBorder">
        <div class="card nobackface frontside">
        FRONT
        </div>
        <div class="card nobackface backside">
        BACK
        </div>
    </div>
</div>
```

例 6-5 展示了文件 css/css3dbackfaces.css 中的样式声明。首先，我们定义了看起来可能有些违反直觉的 `frontside` 和 `backside` 类。`frontside` 用于定义卡片的正面，不过这个示例是为了说明背面的渲染规则，所以我们让卡片绕 *y* 轴旋转 210 度来使得其背面朝向观察者。相对，卡片的背面相对观察者有一个 30 度的旋角。由于旋转角度恰好相差 180 度，所以卡片的双面看起来是叠在一起的。结合用于隐藏背面的 `nobackface` 类，我们得到了一个如右下卡片的完美双面卡片。`nobackface` 类将 `backface-visibility` 属性设置为 `hidden`，以得到期望的效果。

例 6-5：用于创建双面物体的 CSS 声明

```
.frontside {
    -webkit-transform: rotateY(210deg);
       -moz-transform: rotateY(210deg);
         -o-transform: rotateY(210deg);
            transform: rotateY(210deg);
            line-height:160px;
            font-size:40px;
            color:White;
            background-color:DarkCyan;
            border-color:Black;
            box-shadow:2px 2px 2px Black;
}

.backside {
    -webkit-transform: rotateY(30deg);
       -moz-transform: rotateY(30deg);
         -o-transform: rotateY(30deg);
            transform: rotateY(30deg);
            line-height:160px;
            font-size:40px;
            color:White;
            background-color:DarkRed;
            border-color:Black;
            box-shadow:2px 2px 2px Black;
            opacity:0.8;
}

.backface {
    -webkit-backface-visibility: visible;
       -moz-backface-visibility: visible;
         -o-backface-visibility: visible;
```

```
            backface-visibility: visible;
}

.nobackface {
    -webkit-backface-visibility: hidden;
       -moz-backface-visibility: hidden;
         -o-backface-visibility: hidden;
            backface-visibility: hidden;
}
```

6.1.5 CSS变换属性汇总

本节涵盖 CSS 提供的用于为 HTML 元素添加 3D 效果的变换属性。表 6-2 对这些属性进行了汇总。

表6-2：CSS变换属性

属　性	描　述
`transform`	使用一个或多个方法来提供变换（见表 6-1）
`transform-origin`	指定所有变换的原点（默认：50%, 50%, 0）
`perspective`	指定用 CSS 距离单位表示的透视深度（默认：0 = 无透视）
`perspective-origin`	指定 *xy* 坐标系中透视的消失点
`transform-style`	指定一个 3D 元素的后代元素是以平面还是 3D 模式渲染
`backface-visibility`	指定一个背对屏幕的 3D 元素是否被渲染

正如我们看到的那样，CSS 变换提供了一个为页面元素添加 3D 效果的强大途径。CSS 变换在结合过渡和动画来创建动态效果的时候会更加强大。

> 本节中的示例得到了 David DeSandro 的著名博客站点“24 Ways”（例如，打动你朋友的 24 种方法）的大力支持。David 慷慨地允许我对他的工作成果进行改编。参考他网站上的示例以及其他文章，了解更多关于 CSS 3D 的信息。

6.2 CSS过渡

CSS 过渡允许属性随时间渐进地变化。CSS 过渡非常类似于我们在上一章中看到过的 Tween.js 的补间。然而，这些效果是浏览器内置的，不需要任何辅助性的 JavaScript 库。CSS 过渡还可以用于多数的 CSS 属性动画：width、position、color、z-index、opacity 等，但这里我们的关注点在于使用 CSS 来实现 3D 属性的动画。

基本的 CSS 过渡语法如下：

```
transition : property-name duration timing-function delay-time;
```

各项属性的含义如下。

- `property-name`

  过渡应用的属性名称，关键字 `all` 表示过渡将应用于全部产生变化的属性，而关键字

none 表示过渡将不应用于任何属性。

- duration

  以秒或毫秒为单位的时间值，指定过渡时长。

- timing-function

  用于过渡动画的补间函数名。它可以是 linear、ease、ease-in、ease-out、ease-in-out 或 cubic-bezier 中的任意一种。

- delay-time

  指定过渡开始之前等待的时间长度（以秒或毫秒为单位）。

transition 实际上是四个独立的 CSS 属性的简写，这四个属性分别是 transition-property、transition-duration、transition-timing-function 和 transition-delay。让我们通过一个示例来看看它们是如何工作的。打开文件 Chapter 6/css3dtransitions.html，如图 6-8 所示。我们可以看到两个卡片。点击其中任意一个可以将其翻转到另一面，这里使用了上一节介绍的双面技术。翻转过渡持续了两秒的时间，伴随轻微的渐入渐出。卡片的颜色也同时由原本的暗青色（DarkCyan）变为金黄色（Goldenrod）。然而，左边的卡片颜色随着翻转过程而变化，而右边的卡片颜色则在翻转结束后才改变。

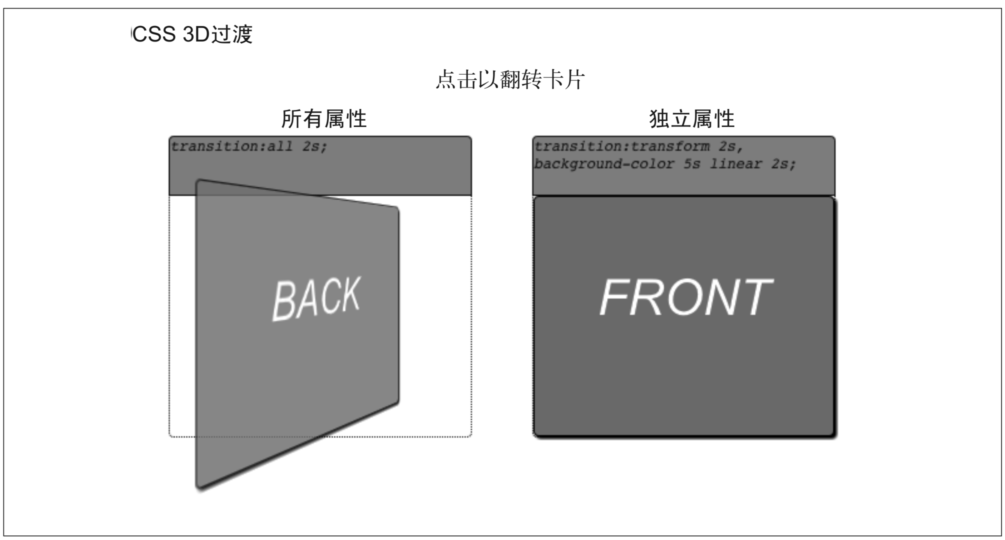


图 6-8：使用 CSS 过渡来进行属性动画（另见彩插图 6-8）

下面的 HTML 代码用类似的方式定义了每个卡片的正面和背面。两个卡片的主要区别在于左边的卡片使用了 CSS 类 easeAll2sec 而右边的卡片使用了 CSS 类 easeTransform2secColor5secDelay。我们稍后将会看到这两个类的代码。

```
<div id="transition1" class="container perspective ">
    <div class="legend">
    All Properties
    </div>
```

```
    <div class="code">transition:all 2s;</div>
    <div class="cardBorder">
        <div id="front1"
          class="card nobackface frontside clickable easeAll2sec">
        FRONT
        </div>
        <div id="back1"
          class="card nobackface backside clickable easeAll2sec">
        BACK
        </div>
    </div>
</div>

<div id="transition2" class="container perspective ">
    <div class="legend">
    Individual Properties
    </div>
    <div class="code">transition:transform 2s,
          background-color 5s linear 2s;</div>
    <div class="cardBorder">
        <div id="front2"
class="card nobackface frontside clickable easeTransform2secColor5secDelay">
        FRONT
        </div>
        <div id="back2"
class="card nobackface backside clickable easeTransform2secColor5secDelay">
        BACK
        </div>
    </div>
</div>
```

效果由鼠标点击事件来驱动。我们用 jQuery 来为每个卡片的正面和背面添加点击响应处理程序。它使用一个布尔值来追踪卡片当前显示的是哪一面，并根据需要添加或移除 `flip` 和 `goGold` 类。`flip` 使卡片旋转了 180 度，`goGold` 将颜色设置为金黄色（`Goldenrod`）。如果没有设置 CSS 过渡，这些变化会立即生效，但有了过渡之后，它们就会从一个状态随时间平滑地变化到另一个状态。

```
<script type="text/javascript">

    var front1 = true;
    var front2 = true;
    $(document).ready(
            function() {
                $('#transition1 .clickable').click(function(){
                    // alert("Clicked");
                    if (front1)
                    {
                $('#front1').addClass('flip');
                $('#back1').addClass('flip');
                $('#front1').addClass('goGold');
                $('#back1').addClass('goGold');
            }
            else
            {
```

```
                $('#front1').removeClass('flip');
                $('#back1').removeClass('flip');
                $('#front1').removeClass('goGold');
                $('#back1').removeClass('goGold');
            }

            front1 = !front1;
        });

        $('#transition2 .clickable').click(function(){
            if (front2)
            {
                $('#front2').addClass('flip');
                $('#back2').addClass('flip');
                $('#front2').addClass('goGold');
                $('#back2').addClass('goGold');
            }
            else
            {
                $('#front2').removeClass('flip');
                $('#back2').removeClass('flip');
                $('#front2').removeClass('goGold');
                $('#back2').removeClass('goGold');
            }

            front2 = !front2;
        });

      }

    );

</script>
```

这个示例的 CSS 代码可以在文件 css/css3dtransitions.css 中找到，例 6-6 列出了这些代码。

卡片的正面和背面由类 `frontside` 和 `backside` 中定义的不同旋转角度来确定，结合类 `flip`，它们通过旋转 180 度来翻转卡片。`goGold` 类用以变换元素背景颜色为金黄色。用粗体标出的两个类定义了两个不同的过渡。`easeAll2sec` 非常简单：它为所有变化的属性添加了一个包含轻微渐入渐出（使用默认的 `ease` 值）、时长为两秒的过渡。

`easeTransform2secColor5secDelay` 则复杂一些。它实际上包含两个过渡，一个用于变换而另一个用于背景色，它们用逗号隔开。变换的过渡与 `easeAll2Sec` 中的完全相同，是一个包含轻微渐入渐出的两秒时长的过渡。背景颜色的过渡则完全不同：它是一个在两秒钟的停留之后才开始执行的、持续 5 秒的线性颜色插值，使用了过渡属性（`transition`）的第四个参数，delay time。

例 6-6：定义 CSS 过渡

```
.frontside {
    -webkit-transform: rotateY(0deg);
       -moz-transform: rotateY(0deg);
         -o-transform: rotateY(0deg);
            transform: rotateY(0deg);
```

```
...
}

.backside {
    -webkit-transform: rotateY(180deg);
       -moz-transform: rotateY(180deg);
         -o-transform: rotateY(180deg);
            transform: rotateY(180deg);
...
}

.frontside.flip {
    -webkit-transform: rotateY(-180deg);
       -moz-transform: rotateY(-180deg);
         -o-transform: rotateY(-180deg);
            transform: rotateY(-180deg);
}


.backside.flip {
    -webkit-transform: rotateY(0deg);
       -moz-transform: rotateY(0deg);
         -o-transform: rotateY(0deg);
            transform: rotateY(0deg);
}

.goGold {
    background-color:Goldenrod;
}

.easeAll2sec {
    -webkit-transition:all 2s;
       -moz-transition:all 2s;
         -o-transition:all 2s;
            transition:all 2s;
}

.easeTransform2secColor5secDelay {
    -webkit-transition:-webkit-transform 2s, background-color 5s linear 2s;
       -moz-transition:-moz-transform 2s, background-color 5s linear 2s;
         -o-transition:-o-transform 2s, background-color 5s linear 2s;
            transition:transform 2s, background-color 5s linear 2s;
}
```

本节仅仅涉及了 CSS 过渡的一点皮毛。在 Microsoft CSS 开发专家 Kirupa Chinnathambi 的博客上（http://www.kirupa.com/html5/all_about_css_transitions.htm），有一篇关于这个属性的优秀文章。

过渡是一个用来创建效果的简单方法，但仅限于创建简单的单次效果。如果我们想要创建复杂的队列和循环，那么需要使用另一项 CSS3 技术：CSS 动画。

6.3　CSS动画

CSS 动画提供了一个比 CSS 过渡更为全面的动画方案。像上一章介绍的 3D 关键帧动画一样，CSS 动画利用一系列关键帧和属性来控制动画时长、缓动函数、延时以及循环。让我们来看看一些示例。

打开文件 Chapter 6/css3danimations.html。你会看到三个卡片。点击它们来触发不同的动画（图 6-9）。左上的卡片进行了一个简单的围绕 *y* 轴旋转的单次动画。右上的卡片重复进行左右晃动的动画。下方的卡片“飞”起来并向右移动，在移动过程中始终围绕 *y* 轴旋转。

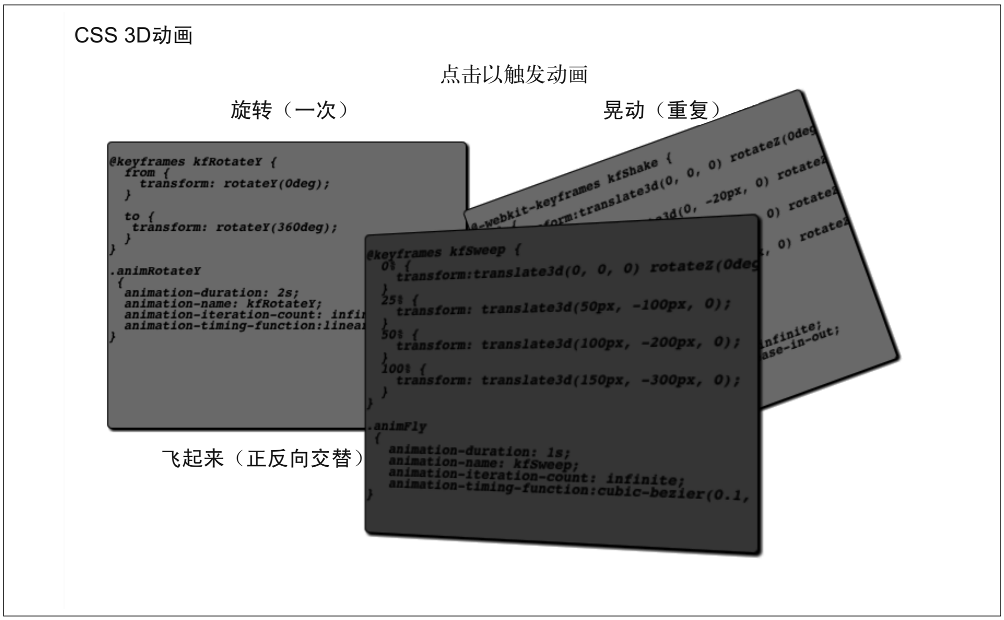


图 6-9：CSS 3D 动画

用于创建动画的 CSS 分为两个部分：一个用于创建关键帧数据的 CSS 代码块的 `@keyframe` 规则，以及可定义于单个元素上的许多属性。

- `animation-name`

  一个由 `@keyframe` 规则定义的关键帧集合的名字，用于作为关键帧数据的源。

- `animation-duration`

  指定动画的时长，以秒或毫秒为单位。

- `animation-timing-function`

  关键帧动画的缓动函数名称，可以是 `linear`、`ease`、`ease-in`、`ease-out`、`ease-in-out` 或 `cubic-bezier` 中的任意一个。

- `animation-delay`

  指定动画开始前等待的一段时间（以秒或毫秒为单位）。

- `animation-iteration-count`

  指定动画播放的次数。默认为 `1`。关键字 `infinite` 用于定义永久循环的动画。

- `animation-direction`

  决定动画是正向、反向还是正反向交替播放。可选的值有 `normal`（正向）、`reverse`（反向）、`alternate`（正反向交替），以及 `alternate-reverse`（正反向交替，先反向播放）。

我们可以用 CSS 属性简写 `animation` 将前面所有的属性像下面这样结合起来：

```
animation: name duration timing-function delay iteration-count direction;
```

图 6-9 对应示例的 CSS 代码可以在文件 css/css3danimations.css 中找到。例 6-7 的摘录展示了其中的重要片段。我们使用 `@keyframe` 规则 `kfRotateY` 和 `kfRotateMinusY` 来设置关键帧（分别用于卡片从正面到背面以及从背面到正面的旋转），`kfShake` 用于晃动的动画，`kfFly` 用于飞翔的动画。我们为每个动画分别定义了不同的类，使用了不同的参数。`animRotateY` 和 `animRotateMinusY` 类定义了让元素绕 y 轴旋转的无限线性插值动画。这些动画都通过简单的表示从开始状态到结束状态的关键帧数据来定义。

`kfShake` 类则更复杂一些：它使用了分别位于 0%、25%、50% 和 100% 时刻的四个关键帧数据，来定义 x 和 y 方向上的平移，以及围绕 z 轴的旋转。最后，`kfFly` 类更为复杂，它在关键帧中定义了一系列平移，使用了一个自定义的三次贝塞尔插值函数，并采用了正向和逆向的交替播放。`kfFly` 仅仅定义了元素的飞行路径；它之所以表现出“拍动翅膀”的视觉效果，是因为元素被点击时添加在元素正面 / 背面的 `animRotateY` 类和 `animRotateMinusY` 类。因此，实际上底部的卡片应用了嵌套动画。

例 6-7：创建关键帧动画的 CSS 声明

```
@-webkit-keyframes kfRotateY {
  from {
    -webkit-transform: rotateY(0deg);
  }

  to {
   -webkit-transform: rotateY(360deg);
  }
}

.animRotateY
 {
  -webkit-animation-duration: 2s;
  -webkit-animation-name: kfRotateY;
  -webkit-animation-iteration-count: infinite;
  -webkit-animation-timing-function:linear;
}

@-webkit-keyframes kfRotateMinusY {
  from {
    -webkit-transform: rotateY(-180deg);
  }

  to {
   -webkit-transform: rotateY(180deg);
```

```
  }
}

.animRotateMinusY
 {
  -webkit-animation-duration: 2s;
  -webkit-animation-name: kfRotateMinusY;
  -webkit-animation-iteration-count: infinite;
  -webkit-animation-timing-function:linear;
}

@-webkit-keyframes kfShake {
  0% {
    -webkit-transform:translate3d(0, 0, 0) rotateZ(0deg);
  }
  25% {
    -webkit-transform: translate3d(0, -20px, 0) rotateZ(20deg);
  }
  50% {
    -webkit-transform: translate3d(0, 0, 0) rotateZ(-20deg);
  }
  100% {
    -webkit-transform: translate3d(0, -20px, 0) rotateZ(-20deg);
  }
}

.animShake
 {
  -webkit-animation-duration: .5s;
  -webkit-animation-name: kfShake;
  -webkit-animation-iteration-count: infinite;
  -webkit-animation-timing-function:ease-in-out;
}

@-webkit-keyframes kfFly {
  0% {
    -webkit-transform:translate3d(0, 0, 0);
  }
  25% {
    -webkit-transform: translate3d(100px, -100px, 20px0);
  }
  50% {
    -webkit-transform: translate3d(200px, -200px, 40px);
  }
  100% {
    -webkit-transform: translate3d(400px, -300px, 20px);
  }
}

.animFly
 {
  -webkit-animation-duration: 2s;
  -webkit-animation-name: kfFly;
  -webkit-animation-iteration-count: 2;
  -webkit-animation-timing-function:cubic-bezier(0.1, 0.2, 0.8, 1);
```

```
  -webkit-animation-direction:alternate;
}
```

你也许已经注意到，animRotateY 和 animRotateMinusY 中的 animation-iteration-count 属性都被定义为 infinite，然而左上的卡片仅仅旋转了一次。这是因为 jQuery 代码在一次播放后停止了动画（用粗体标出的代码）。

```
        $('#front1').click(function(){
            $('#front1').addClass('animRotateY');
            $('#back1').addClass('animRotateMinusY');
            setTimeout(function(){
                $('#front1').removeClass('animRotateY');
                $('#back1').removeClass('animRotateMinusY');

            }, 2000);
            }
        );
```

这些类设计为无限循环，是为了复用在不同的动画效果上，例如它们与 animFly 类结合的时候。通过将它们定义为无限循环，我们可以轻松地在 JavaScript 中控制它们的开始和停止。

6.4　挑战CSS的极限

到目前为止，本章中的示例主要集中于平面物体的移动。然而这些技术同样也可以用来创建优秀的 3D 用户界面元素以及变换效果，但是它们与现今的游戏和其他全 3D 的应用效果之间还有一定的距离。即便如此，一些开发者仍然在试图扩展这项技术的边界。本章剩下的部分将讲述我们究竟可以用 CSS3 创建出什么样的 3D 效果。

6.4.1　渲染3D物体

在前面的几节中我们看到，为了创建双面的扁平物体，我们编写了一些 HTML 和 CSS 代码。为了创建带深度的物体，如立方体，我们将耗费更多的精力。许多 CSS3 相关的站点展示了关于如何实现这些效果的优秀示例。图 6-10 描绘了一个 3D 的虚拟产品盒子，这个作品由德国开发者 Dirk Weber 为他的 HTML5 开发站点 http://www.eleqtriq.com 创建。这个盒子具有前、后、两侧、顶和底，并可以旋转。它甚至有虚拟的倒影！



图 6-10：用 CSS 创建的可旋转 3D 物体（http://www.eleqtriq.com/2010/11/natural-object-rotation-with-css3-3d/）

Codrops（一个 Web 设计和开发博客，http://tympanus.net/codrops/）团队进一步拓展了这个盒子的概念，他们创建了一个 3D 的虚拟书籍展示。你可以打开封面来阅读书中的内容，还可以翻页。如图 6-11 所示。

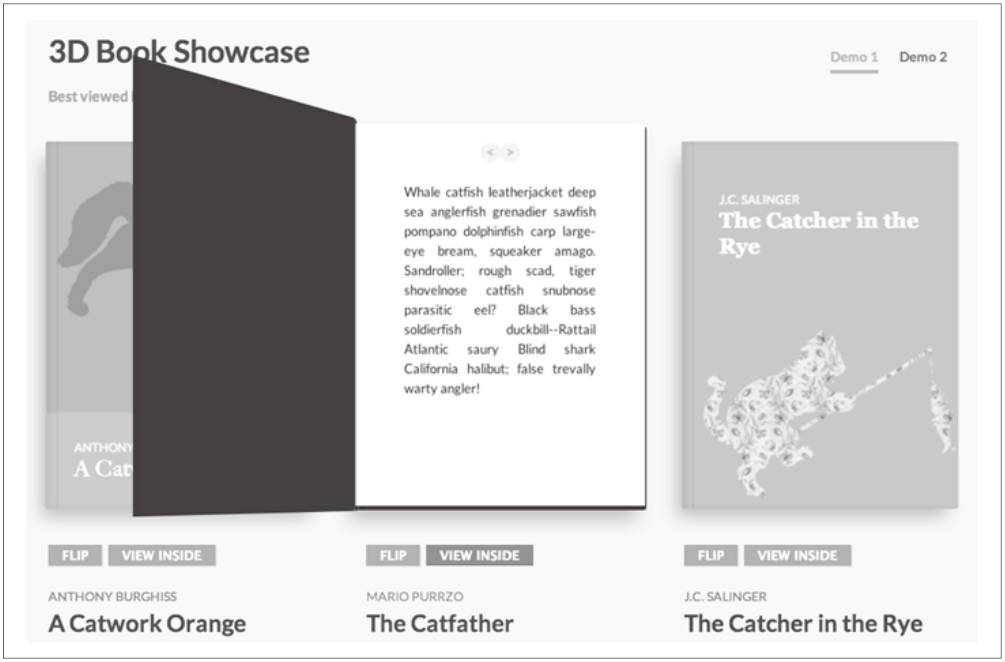


图 6-11：3D 虚拟书籍展示（http://tympanus.net/codrops/2013/01/08/3d-book-showcase/）

创建这类完整的 3D 物体需要为每个面创建一个或多个 HTML 元素，并定义许多 CSS 类，通常还需要一些用于逻辑控制的 JavaScript。它非常困难，但这些努力的结果是值得的。看看这里和附录中罗列的 CSS 3D 站点。这些站点中的多数站点都自由分享它们的源代码，这为我们的 CSS 3D 创作提供了一个很好的起点。

6.4.2　渲染3D环境

鉴于 CSS 3D 基本上是基于对矩形物体的操作来实现的，它看起来不太可能被用来创建身临其境的游戏环境。而令人难以置信的是，英国开发者 Keith Clark 做到了，他使用 JavaScript 和 CSS 3D 变换，创建了一个暗黑风格的第一人称射击演示。结果如图 6-12 所示。



图 6-12：用 CSS 3D 和 JavaScript 构建的第一人称射击演示（http://blog.keithclark.co.uk/creating-3d-worlds-with-html-and-css/）

Clark 的作品展示了 CSS3 看起来不可能实现的一些特性，甚至还包括 3D 变换。

- 3D 几何体

  CSS 仅仅可以操作矩形。但是当我们意识到真正的 3D 渲染系统通常也是基于将多个平面多边形——通常是三角形或者四边形——组合成更复杂的形状来操作时，这一点并不能称为局限。同样，我们可以利用 PNG 图片来作为单个四边形中的 alpha 通道遮罩，从而创建出具有各种剪切形状的多边形。有了这两个特性，Clark 就可以利用 CSS 来创建圆桶、枪支、射击者的手部，以及其他仿真的 3D 几何体。

- 相机、导航和碰撞

  CSS 支持基本的透视，但 Clark 想出了如何根据键盘输入实时移动一个虚拟的相机，同时通过将玩家的位置投影到 2D 水平面上，并与一个手工制作的 2D 高度图进行比较来计算碰撞。

- 灯光和阴影

  CSS 不支持元素的照明。为了创建仿真的光照模型，Clark 需要构建每个四边形的法向量，在 JavaScript 中创建虚拟的光源，在一个 `<canvas>` 元素中离屏渲染纹理映射，将其和基础纹理映射混合，来创建出一个被光线照亮的表面。

许多开发者并不愿意冒险去像 Keith Clark 一样进行那么复杂的工作。像这样的环境如果使用 WebGL 和类似 Three.js 这样的库来搭建，会非常简单。尽管如此，这个项目仍然是用于展现 CSS3 能力的、非常优秀的案例研究。如果希望了解更多信息，请参见 Keith Clark 对这个项目的说明（http://keithclark.co.uk/articles/creating-3d-worlds-with-html-and-css/）。

6.4.3　使用CSS自定义滤镜来实现高级着色器效果

某些浏览器试验性地允许开发者使用 GLSL 着色器语言来将任意的 3D 效果作用于 CSS 元素。这项由 Adobe 系统开拓的技术被称为 CSS 自定义滤镜（旧称为 CSS 着色器）。图 6-13 展示了在添加 CSS 自定义滤镜“揉皱”效果之前和之后的 DOM 元素的效果。当鼠标在元素上移动时，着色器程序扭曲了组成元素显示矩形的节点，使顶点在短时间内产生运动，直到它们看起来像一张皱巴巴的纸。关于使用 CSS 自定义滤镜，最值得注意的地方在于，这个 DOM 元素的内容是标准的 HTML：一些带样式的文字，以及一张图像。CSS 自定义滤镜允许 Web 开发者充分利用现有的 HTML 知识，创建新的吸引眼球的交互效果。

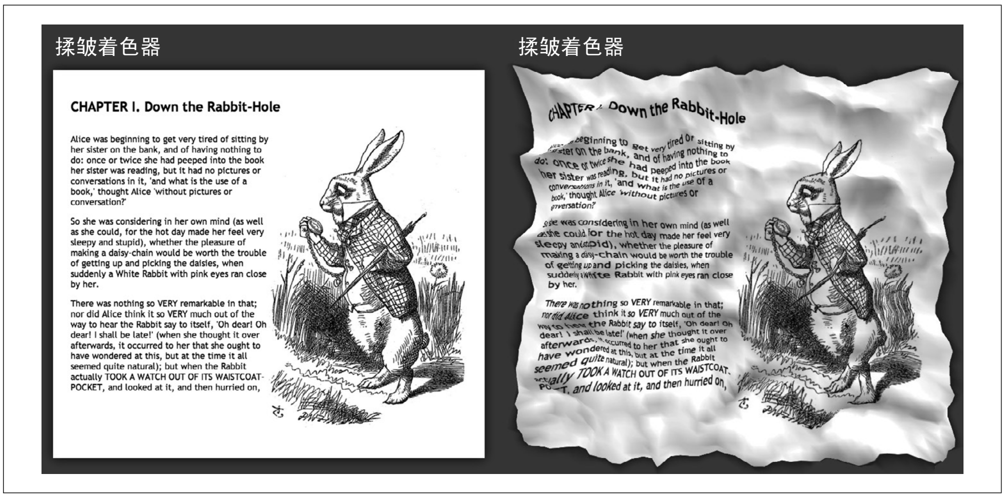


图 6-13：揉皱着色器，一个 CSS3 自定义滤镜；由 Altered Qualia 提供（http://alteredqualia.com/css-shaders/crumple.html/）

CSS 自定义滤镜使用了 GLSL 着色器语言的一个子集（GLSL ES）。它基本上和 WebGL 中使用的 GLSL ES 相同，但有一些非常细微的差异。出于安全性考虑，不允许 CSS 自定义滤镜直接访问页面元素上的像素颜色；相反，滤镜需要计算出一个混合色彩，它最终将作用于目标像素，用于计算出最终的颜色。此外，浏览器还以全局变量的形式提供了一些预定义的值，如由元素的标准 CSS 3D 属性（之前我们讨论过）定义的 3D 变换矩阵。另一个重要的差异是 CSS 自定义滤镜是可选的，而在 WebGL 渲染中着色器则是必需的。

> 注意 CSS 自定义滤镜还是一个实验性的特性，它仅仅被某些浏览器支持。在本书写作时，这个特性还有被搁置的危险，因为某些人更倾向于将 DOM 元素作为纹理映射整合到 WebGL 标准中，而这项工作也取得了相当的进展。在此期间这项特性仍被 Chrome 所支持，你可以通过一个特殊的命令行（`--enable-css-shaders`）来开启它，或者在用户偏好设置中启用 CSS 着色器。

6.4.4 用Three.js来渲染CSS 3D

即便在 2014 年，许多浏览器仍然不支持 WebGL，包括 iOS 上的 Mobile Safari。因此许多时候我们需要使用降级的技术来创建 3D。正如之前我们已经看到的那样，CSS3 是其中一个选择。然而基于 CSS 来进行深度的 3D 开发需要很多付出劳动，往往为了创建几个 3D 对象，就需要大量的 CSS 类和 HTML 元素。

最近，Mr.doob 开始着手为 Three.js 创建一个基于 CSS 的渲染系统。Three.js 最出色的一点在于它可以使用不同的浏览器技术来进行渲染。Three.js 的渲染架构是插件式的，它提供了内置的 WebGL、2D Canvas、SVG 以及 CSS 渲染器。

Three.js 的 CSS 渲染器使用 CSS 变换来对物体进行平移、旋转以及缩放，它是将交互式页面元素映射到 3D 空间中的理想方案。如图 6-14 中描绘了一个交互式的元素周期表。表格的每个单元格都是一个具备完整功能的 DIV 标签，因此它可以用 HTML 来渲染并使用 CSS 来添加样式。CSS 渲染器是为矩形、富文本物体创建创新性布局的优秀选择。

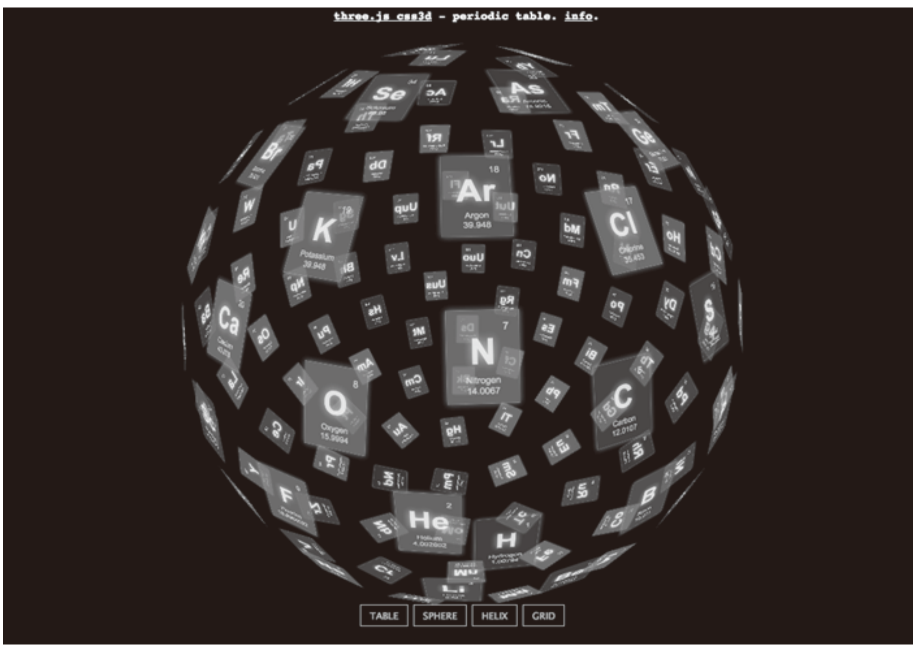


图 6-14：一个交互式的元素周期表，用 Three.js 创建，使用 CSS 3D 变换渲染（http://mrdoob.github.io/three.js/examples/css3d_periodictable.html）

6.5 小结

本章研究了用于创建 3D 效果的浏览器内置 CSS3 属性：CSS 变换、CSS 过渡以及 CSS 动画。你了解到了如何使用 CSS 变换来将 3D 平移、旋转和缩放应用于元素，并使用或不使

用透视来渲染它们；通过 DOM 层级结构来传播 3D 变换；控制一个元素的背面渲染。我们使用 CSS 过渡创建了简单的动画效果，并使用 CSS 动画创建了更复杂的效果。

CSS3 提供了用于创建 3D 用户界面元素和变换效果的强大能力，但它并不适合用来构建现今游戏和其他图形密集 3D 应用的效果。但从另一方面来说，这些效果非常容易创建；它们可以通过 CSS 辅以少量 JavaScript 代码的方式来实现；它们是跨平台、跨浏览器的；最重要的是，它们是浏览器内置的特性，这表示你无需为实现这些效果引入额外的库。在极少数情况下，当想要挑战 CSS3 的极限时，我们可以使用大量的 JavaScript 技巧或引入类似 Three.js 这样可以用 CSS3 来渲染的库。

第 7 章

Canvas：通用2D绘图

最终，3D 图形要绘制到 2D 的平面设备上，比如电脑、平板以及手机。深度和透视效果使得它们看起来像 3D，使得某些物体看起来比较近、某些看起来比较远。如果我们希望 3D 内容是交互式的，那么渲染速度就需要足够快，至少每秒 30 帧，最好每秒 60 帧。

WebGL 和 CSS3 使用 GPU（现代计算机和其他设备中专门用来处理图形的硬件）来实现 3D 实时渲染。尽管 3D 硬件加速对于交互式 3D 图形来说极其重要，但它并不是必需的。使用软件渲染同样可以创造出令人惊叹的 3D 体验。对于 Web 应用来说，软件渲染意味着使用 Canvas 2D，一个在浏览器中常见的 2D 图形绘制 API。

在某些场景下，我们需要考虑使用 Canvas 2D 而不是 WebGL。一种场景是，尽管 WebGL 很普及，但写作本书时它并没有被所有移动平台所支持，尤其是 iOS 的 Safari 中[1]。在这些平台上，我们可以使用 Canvas 2D 作为降级方案，尽管这么做可能会影响性能或牺牲画质。另一种场景是，我们可能需要支持电量有限的设备，比如某些智能手机的 GPU 非常耗电，所以我们需要使用纯软件渲染来延长续航时间。还有一种场景是，我们想要创建一个简单的 3D 效果，使用 WebGL 有点杀鸡用牛刀，而 CSS 又不够强大。这些场景更适合使用 Canvas 2D 来取代 WebGL。

在本章中，我们将研究如何使用 Canvas API 来渲染 3D，以及如何权衡性能和效果。我们还会研究一些处理 3D 数学和渲染的开源库，这些库能让我们专注于开发应用。

7.1 Canvas基础

苹果在 2004 年首先引入 Canvas，为了在 Dashboard 组件和 Safari 浏览器中支持高级界面的开发。它的做法是提供一个绘制图形的通用画布。在接下来的几年中，它被 Mozilla 的

注 1：iOS 8 中已经支持。——译者注

Gecko 引擎、其他基于 WebKit 引擎的浏览器（如 Chrome）所支持，最终普及到所有支持 HTML5 的浏览器和平台。

和之前 Web 标准中绘制 2D 矢量图形的 DOM 元素及 SVG 不同，Canvas 并没有在标签中提供一系列固定的形状元素，它提供了一个 API 来让 JavaScript 开发者任意绘制和填充各种图形，包括直线、曲线、多边形和文本。而且不同于 DOM 和 SVG，Canvas 使用了类似 WebGL 的过程模型。在浏览器的场景图中并没有包含 Canvas 元素内部的内容，所以应用必须维护好它内部的对象和绘制操作，以便在重绘的时候使用（比如在动画中）。

全面研究 Canvas API 超出了本书的范畴，但为了理解和 3D 相关的 Canvas 绘图，我们需要简单介绍一下它的基础知识。

7.1.1 Canvas元素和2D绘图上下文

HTML5 定义了一个新的 DOM 元素：`<Canvas>`，它将页面中的一个区域定义为可绘制的。Canvas 元素类似于图像元素：你可以使用标签创建它，或者使用 DOM 中的 `document.createElement()` API。当创建好后，你可以使用 CSS 来控制 Canvas 元素的样式，比如设置边框、外边距、位置等，甚至使用过渡添加动画效果。

Canvas 元素仅仅声明了页面中的某个区域用来绘图。在绘图的时候，你需要通过上下文（context）对象绘制，这个对象提供了绘图的 API。对于 Canvas 绘图，我们需要使用 2D 的上下文对象，而不是在前几章中用于 WebGL 绘图的 3D 上下文对象。

例 7-1 展示了如何创建一个 Canvas 元素并绘制一个白色正方形。在 `style` 标签中，我们为 Canvas 元素指定了黑色的背景。在 `<body>` 标签中，我们创建了一个以像素为单位指定高宽的 `<canvas>` 标签。在页面 domready 的时候，我们通过 `id` 取得 Canvas 元素，然后通过 `canvas.getContext("2d")` 获得 2D 上下文对象。当获得上下文后，我们将它的 `fillStyle` 属性设置为白色，然后调用 `context.filleRect()`，传入 x、y 作为左上角的坐标点，以及高和宽，来填充一个矩形区域。

例 7-1：基本 Canvas 绘图示例

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Programming 3D Applications in HTML5 and WebGL &mdash;
       Basic Canvas Example</title>

</head>

<style>
    #basicCanvas {
        background-color:Black;
    }
</style>
<body>

<canvas id="basicCanvas" width=500 height=500></canvas>

</body>
```

```
<script src="../libs/jquery-1.9.1/jquery-1.9.1.js"></script>
<script type="text/javascript">

    $(document).ready(
            function() {

                var canvas = document.getElementById("basicCanvas");
                var context = canvas.getContext("2d");
                context.fillStyle = '#ffffff';
                context.fillRect(125, 125, 250, 250);
            }
    );

</script>
</html>
```

显示结果很眼熟，请看图 7-1。

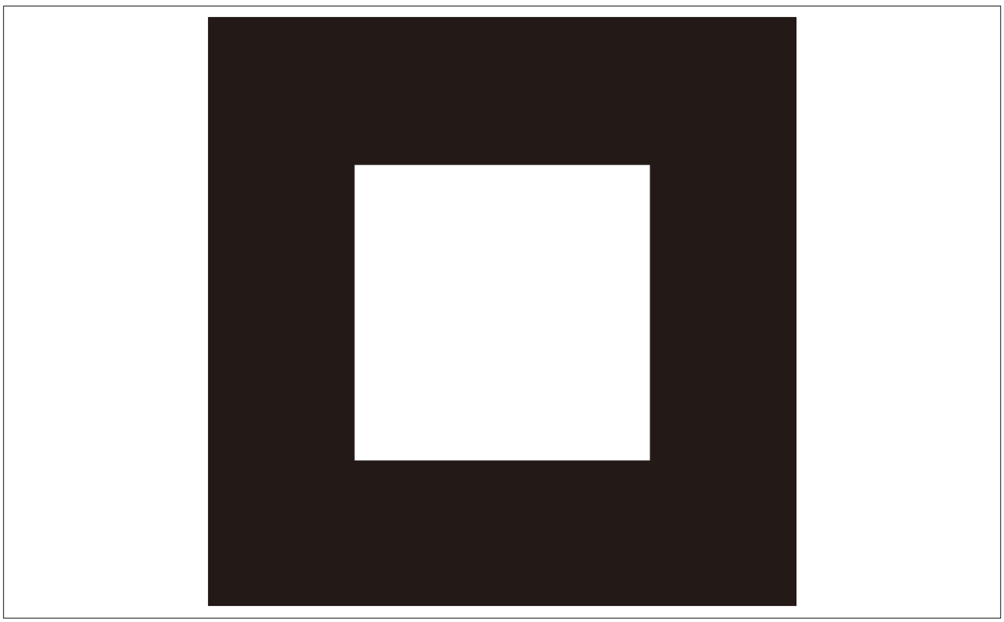


图 7-1：使用 Canvas API 绘制一个正方形

十分简单吧。它与第 2 章和第 3 章的例子看起来很像，然而它只用了几行 JavaScript 代码（我之前说过在页面中绘制 2D 图形还有更简单的方法，我不是开玩笑的）。这个例子的代码可以在 Chapter 7/canvasbasic.html 中找到。

7.1.2　Canvas API的功能

Canvas 2D 上下文提供了一个基于光栅的 API。也就是说，绘制是基于像素的（而不是其他图形系统中的矢量图，如 SVG）。如果应用需要基于窗口大小缩放图形，它需要手动处理。2D Canvas 的 API 调用可以分为以下几种。

- 形状绘制

  矩形、多边形以及曲线形状，可以是填充的或只有边框。
- 直线和路径绘制

  线段、弧线和贝塞尔曲线。
- 图像绘制

  来自图像元素或其他 Canvas 等的位图数据。
- 文本绘制

  填充或描边文本，使用根据 CSS 样式定义的文本属性来控制文本。
- 填充或描边

  使用 CSS 样式和渐变等来填充或描边。
- 变换

  2D 变换，包括平移、旋转、缩放以及任意的 3×3 变换矩阵。
- 合成

  控制新绘制的图形和已有的 Canvas 内容如何混合。

图 7-2 显示了调用各种 Canvas API 功能后的效果。我们可以看到一个填充的矩形、一个描边的矩形、填充及描边的文本、一个填充的多边形（三角形）、一个填充且描边的贝塞尔曲线、一个位图、一个使用位图来进行平铺的圆形、一条折线，以及一个渐变填充的矩形。

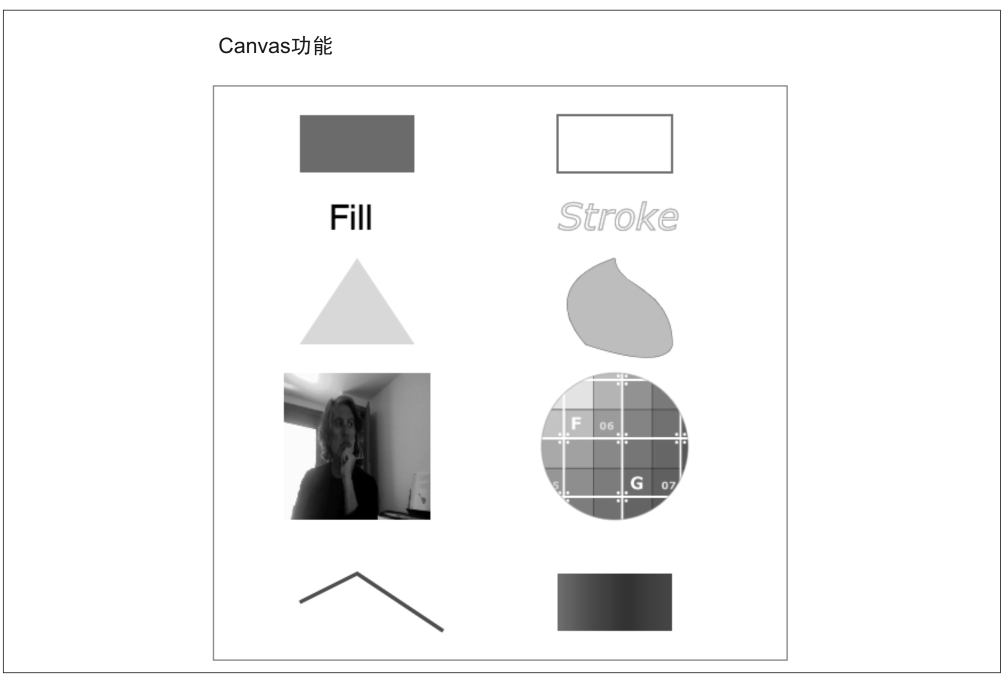


图 7-2：Canvas API 的功能（另见彩插图 7-2）

例 7-2 展示了这个例子中的 JavaScript 代码（文件在 Chapter 7/canvasfeatures.html），为了简化我们省略了其他标签部分。

例 7-2：详细的 Canvas 绘图示例

```
function init()
{
    image1 = new Image;
    image1.src = '../images/parisi1.jpg';

    image2 = new Image;
    image2.onload = function()
    {
        imagepattern = context.createPattern(image2, "repeat");
    }

    image2.src = '../images/ash_uvgrid01.jpg';

    gradient = context.createLinearGradient(250,0,350,0);
    gradient.addColorStop(0,"green");
    gradient.addColorStop(1,"blue");
}

function run()
{
    requestAnimationFrame(run);
    draw(canvas, context);
}

$(document).ready(
        function() {

            canvas = document.getElementById("features");
            context = canvas.getContext("2d");
            init();
            run();
        }
);
```

首先，在页面 domready 的时候查找 Canvas 元素并获得 2D 上下文对象，然后调用 `init()`，最后调用 `run()`。`run()` 基于 `requestAnimationFrame()` 来实现运行循环。和之前的只绘制一次的例子不同，这次我们不断进行重绘。我们这么做有两个原因：(1) 这是更符合真实 Canvas 应用的结构，因为内容某些时候会有动画或相应的用户操作；(2) 这里在某些帧中需要这么做，因为我们需要检测图像是否已经加载完毕。我们需要等到图像加载完毕后才绘制，具体细节待会将进行介绍。

`init()` 函数创建了两个图像元素，一个作为位图内容，另一个用作渐变效果，在右下角的矩形中进行平铺。它还基于第二个位图创建了一个填充效果，为此，在加载图像前添加了一个 `onload` 事件处理器。`onload` 使用上下文的 `createPattern()` 函数来创建平铺。`createPattern()` 方法需要有效的位图数据，所以我们必须等到图像加载完成才能调用。

`run()` 函数实现了运行循环。首先，它让浏览器在下次更新周期的时候再调用自己；接着，它调用 `draw()` 来实现绘图。`draw()` 的代码如下所示：

```
function draw(canvas, context)
{
    context.clearRect(0, 0, canvas.width, canvas.height);

    context.save();
    context.translate(50, 0);

    // 红色填充的小矩形
    context.save();
    context.fillStyle = '#ff0000';
    context.fillRect(25, 25, 100, 50);
    context.restore();

    // 深蓝色描边的小矩形
    context.save();
    context.strokeStyle = 'DarkBlue';
    context.strokeRect(250, 25, 100, 50);
    context.restore();

    // 填充文本
    context.save();
    context.lineWidth = 1;
    context.fillStyle = 'Black';
    context.font = '30px sans-serif';
    context.fillText('Fill', 50, 125);
    context.restore();

    // 对文字进行描边
    context.save();
    context.lineWidth = 1;
    context.strokeStyle = 'Orange';
    context.font = 'italic 2em Verdana';
    context.strokeText('Stroke', 250, 125);
    context.restore();

    // 一个三角形
    context.save();
    context.beginPath();
    context.fillStyle = 'Yellow';
    context.moveTo(75, 150);
    context.lineTo(25, 225);
    context.lineTo(125, 225);
    context.lineTo(75, 150);
    context.fill();
    context.closePath();
    context.restore();

    // 一条填充的贝塞尔曲线
    context.save();
    context.beginPath();
    context.strokeStyle = 'Green';
    context.fillStyle = 'LightBlue';
    context.moveTo(300,150);
    context.bezierCurveTo(225,175,275,225,275,225);
    context.bezierCurveTo(350,250,350,225,350,225);
```

```
        context.bezierCurveTo(350,175,300,175,300,150);
        context.stroke();
        context.fill();
        context.closePath();
        context.restore();

        // 一个位图
        if (image1.width)
        {
            context.save();
            context.drawImage(image1, 11, 250, 128, 128);
            context.restore();
        }

        // 一个用位图填充的圆
        if (image2.width)
        {
            context.save();
            context.strokeStyle = 'DarkGray';
            context.fillStyle = imagepattern;
            context.beginPath();
            context.arc(300, 314, 64, 0, 2 * Math.PI, false);
            context.scale(.5, .5);
            context.fill();
            context.stroke();
            context.closePath();
            context.restore();
        }

        // 一条折线
        context.save();
        context.strokeStyle = "rgb(128, 0, 255)";
        context.beginPath();
        context.lineWidth = 3;
        context.moveTo(25, 450);
        context.lineTo(75, 425);
        context.lineTo(150, 475);
        context.stroke();
        context.closePath();
        context.restore();

        // 一个渐变填充
        context.save();
        context.fillStyle = gradient;
        context.fillRect(250, 425, 100, 50);
        context.restore();

        context.restore();
    }
```

draw() 函数显示了 Canvas 2D API 的许多特性，这里我将强调以下几点。

- context.clearRect() 被用来清除 canvas 中的内容。如果没有它，图形会不断在之前的帧上叠加。
- context.translate() 被用来移动所有随后绘制的物体的位置，它的值会添加到所有其

他绘图操作中。

- 注意到 `context.save()` 和 `context.restore()` 的使用。这两个方法允许开发者在修改绘图状态前保存状态，并在绘制结束后恢复。它会保存变换、描边和填充样式、字体、线段宽度等状态。浏览器使用一个栈来保存这些状态，所以这些方法都是可以被内嵌调用的。它对于绘制具有层次结构的物体非常有帮助。一般情况下，我们不希望一个绘图操作影响另一个绘图操作。然而，需要注意这个操作会带来一定的性能开销，所以在使用它们的时候需要小心。
- `beginPath()` 和 `closePath()` 方法允许我们自定义折线和曲线的路径。Canvas 维护了一个虚拟的笔位置，我们通过 `moveTo()`、`lineTo()` 和 `bezierCurveTo()` 这些方法来控制它。`beginPath()` 方法会重置笔的状态，而 `closePath()` 方法则将当前笔的位置和最开始调用 `moveTo()` 时笔的位置连起来。
- 图像的绘制是通过 `context.drawImage()` 来完成的。我们需要等待图像加载完后才能绘制它，而这通过测试图像的宽度不为零来判断。`drawImage()` 函数可以以图像本身的大小来进行绘制，或者通过传入第四和第五个参数来控制它的宽高，从而进行缩放。图像同样可以被用来平铺物体，我们等 `image2` 加载完后调用 `context.createPattern()` 来创建平铺效果。通过它生成的平铺对象会保存到 `imagepattern` 变量中,然后用来平铺圆形。

这个例子只涉及 Canvas 2D 图形绘制的部分 API，它还有其他大量的功能。使用 Canvas 渲染有它自己的性能问题及最佳实践，但它已经超出了本书的范畴。请参考本书附录来了解 Canvas API 的其他信息。

7.2 使用Canvas API来渲染3D

前面我们学习了 Canvas 2D API 的基础知识，现在我们可以开始讨论使用它来渲染 3D 系统时的问题。尽管有多种方法，但大部分软件渲染的实现都模仿了硬件渲染流程，先将带颜色的三角形、线条、点从模型（对象）空间变换到屏幕空间，然后再进行绘制。

使用 3D 硬件加速的时候，我们用 GLSL 着色器代码来进行计算，它有强大的内建函数，并且编译为在 GPU 中执行的底层机器码。但如果没有 3D 加速硬件，我们就需要先在 JavaScript 中实现这些计算，然后再把上好颜色且变换好后的物体通过 Canvas API 进行最终渲染。这些计算包括维护 3D 几何体、变换、光线、着色，以及将 3D 物体投影到 2D 视口上。上述工作需要大量的计算，即便在高性能的机器上也有很大负担，而开发者所耗费的脑力就更用不提了。

软件渲染器通常会做以下几个工作。

- 变换三角形，将三角形从物体空间变换到屏幕空间。这需要乘以几个矩阵，具体视场景的复杂度而定。在最低限度上，它需要将物体的三角形从世界空间（假设没有其他更多的变换）变换到相机空间，然后通过透视投影将它变换到屏幕的 2D 空间。
- 将三角形进行着色，这需要根据具体的材质来定。如果涉及光照，顶点法向量和光源都需要纳入进来。使用 Canvas 2D API 需要动态生成纹理或渐变来创建光线效果，这会非常耗 CPU。如果材质有纹理，材质还需要进行过滤、透视修正以及其他操作来让它看起来平滑和逼真。尤其是很难实现实时修正透视和过滤。你将会在下面的例子中发现，

应用中的纹理差异非常大。

- 将三角形进行排序，基于它和视口的距离。为了让我们的场景看起来正确，靠视口前面的三角形应该渲染在前面，也就是说要遮住后面的三角形。在硬件加速系统中使用深度缓存（depth buffer，也被称为 z 缓存）来实现，它跟踪每个像素点与相机的距离。深度缓存和颜色缓存是并列的。在每个坐标上有颜色缓存，同时也有深度缓存来记录这个点离相机的距离，渲染器会在绘制像素点到颜色缓存的时候测试它。如果这个像素点比之前在深度缓存中记录的位置更近，就绘制它；如果不是，就不绘制。软件渲染几乎都没有深度缓存，因为它太占内存了，而且计算成本很高。作为替代，它们采用三角形排序的方法，通过矩形某个点的位置来进行判断。这个点通常会选择三角形的集合中心点，不过也可以使用最近或最远的 *z* 值，这里并没有标准的做法。对三角形进行排序是软件渲染中最影响性能的部分，你会发现整体矩形的数量决定了你的性能。

即使一个高质量的软件渲染实现解决了所有阻挡在它前面的问题，反锯齿渲染（对物体的边缘进行平滑处理）还是很难通过软件实时做到的，它需要多次渲染一个物体或整个场景。使用 mipmapping 和双向性过滤（bilinear filtering）来对材质进行过滤的计算成本非常高，这导致软件渲染的纹理看起来粗糙且颗粒状明显。请看图 7-3 的例子。而且，位图填充的速度（例如，对 CSS sprite 来说）在软件渲染中远比硬件中慢。

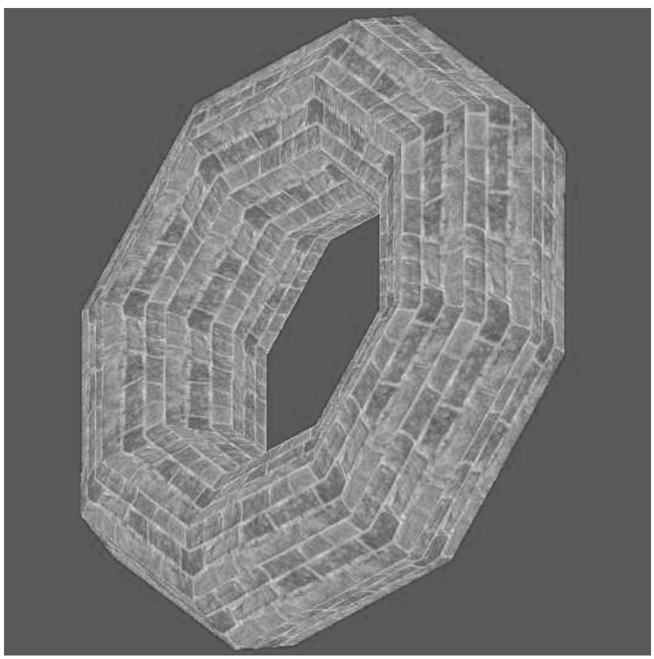


图 7-3：基于软件渲染的纹理映射；转载已被许可

另外，三角形排序虽然在有些场景下可以替代深度缓存，但在其他场景下会有明显问题。例如，当两个三角形重叠的时候，没有好的方法对它们进行排序。请看图 7-4，从相机的视

角看，三角形 B 既在三角形 A 的前面也在三角形 A 的后面。一个软件渲染排序算法必须选择哪个三角形在前面，以此完全遮住另一个三角形。结果就是，当物体相对相机移动的时候，你会偶尔看到三角形跳出来或被隐藏，这在使用深度缓存的硬件渲染中永远不会发生。

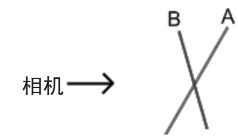


图 7-4：三角形的深度排序——三角形 A 的一部分比三角形 B 更靠近相机，但三角形 B 同样也有一部分比三角形 A 更靠近相机，所以没有好的解决方案（图像来自 MSDN 关于深度排序的文章 http://blogs.msdn.com/b/shawnhar/archive/2009/02/18/depth-sorting-alpha-blended-objects.aspx；转载已被许可）

我们可以看到，在能用硬件渲染的时候使用软件渲染并不是一个理想的方案。获得相同的视觉效果及性能即使有可能，也非常困难。然而尽管有许多内在的限制，但依然有些令人惊叹的努力去使用 2D Canvas API 来创建高性能 3D 效果。

几年前，英国的 Jean dArc 创建了一个浏览 *Quake 3* 关卡地图的查看器，它是基于 2D Canvas 的。图 7-5 显示了它的截图。你可以访问 http://www.zynaps.com/site/experiments/quake.html 来尝试。性能在较新的笔记本上可以接受，尽管纹理因为没有过滤看起来颗粒状明显，但还是很不错的。这之前是 Chrome 实验中用于展示 2D Canvas 能力的例子，在它开发的时候 WebGL 还远不流行。尽管现在它的重要意义已经成为历史，但它展示了 Canvas 能做的事情和软件渲染技术的强大。

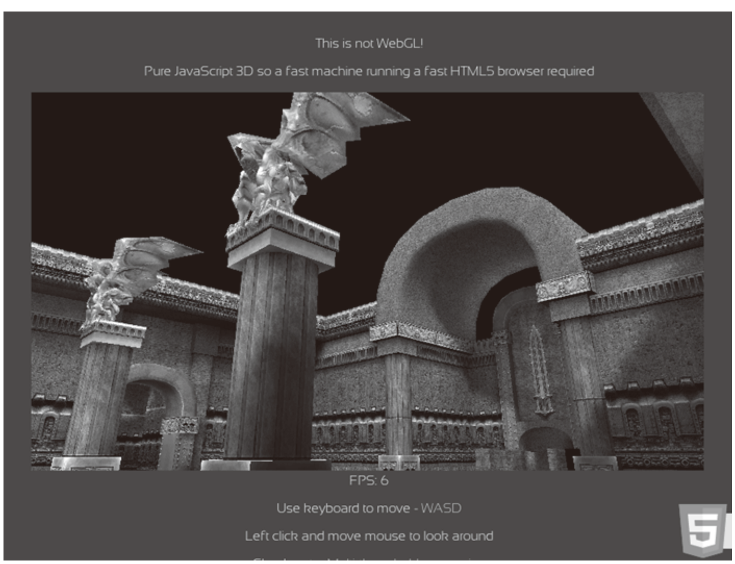


图 7-5：使用 2D Canvas API 渲染的 *Quake 3* 地图查看器（http://www.zynaps.com/site/experiments/quake.html）；转载已被许可

7.3 基于Canvas渲染的3D库

如之前讨论的那样，使用 Canvas 来渲染 3D 有许多技术问题。为了尝试解决它，出现了几个库，包括 K3D（https://launchpad.net/canvask3d）、Cango3D（http://arc.id.au/Canvas3DGraphics.html）、Nihilogic（http://www.nihilogic.dk/labs/canvas3d/）以及 Three.js（https://github.com/mrdoob/three.js/）。在本节中，我们将研究它们中的两个：K3D 和 Three.js。

7.3.1 K3D

K3D 由英国的 Kevin Roast（http://www.kevs3d.co.uk/dev/；Twitter 账号 @kevinroast）所创建。Kevin 是一个 UI 开发者和图形爱好者。尽管 K3D 还在开发早期，没有 Three.js 那样丰富的功能，但它令人印象深刻。具体来说，它性能很好，对于着色与纹理的支持不错。图 7-6 展示了 *Asteroids [Reloaded]* 的截屏，它是一款基于 K3D 实现的经典游戏。注意到岩石上平滑的着色、光线，以及高精细的纹理。

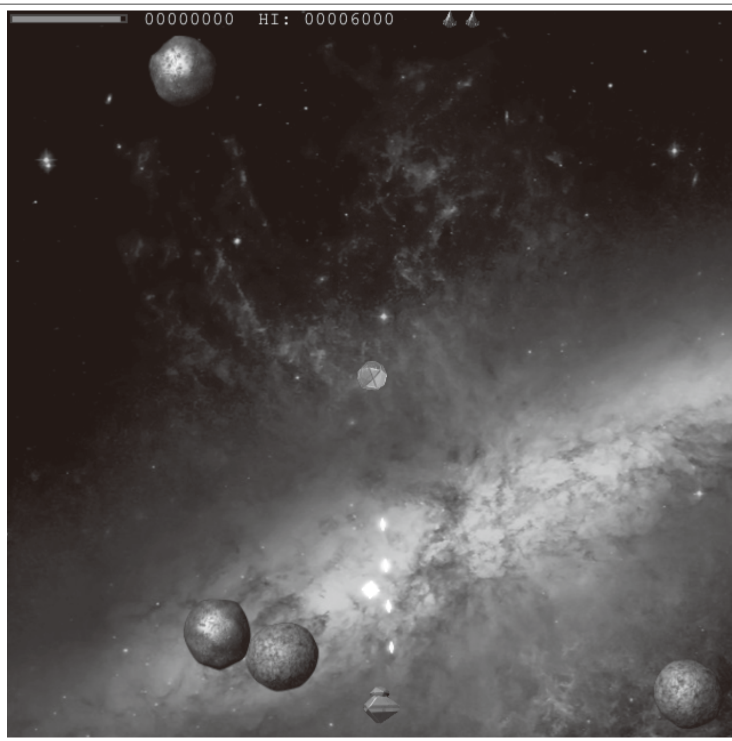


图 7-6：*Asteroids [Reloaded]*，一款基于 K3D 的 3D 游戏，使用 2D Canvas API 进行绘制（http://www.kevs3d.co.uk/dev/asteroids/）

基于之前在 K3D 上的工作，Kevin 现在对 K3D 进行了重写，开发了 Phoria（http://www.kevs3d.co.uk/dev/phoria/）。Phoria 承诺会提供更强大的功能及更为通用，但它依然处于开发早期，目前它的例子还远不够吸引人。

7.3.2　Three.js的Canvas渲染

既然我们使用 Three.js 来开发本书中的其他例子，考虑使用它来做基于软件的 3D 渲染也是很自然的，尤其是如果主要目标是对不支持 WebGL 的平台开发降级方案。使用 Three.js，我们可以在有条件使用 WebGL 的时候使用 WebGL，在没条件的时候使用 Canvas，而这不需要修改太多代码。尽管从 3D 渲染切换到 2D 渲染并不是完全透明的，你需要写几行代码来充分使用 Canvas 渲染的功能，但这些代码不足挂齿。

> Three.js 使用了渲染插件的架构，并自带了可以直接使用的 2D Canvas 渲染器。这并不令人惊奇，因为 Three.js 最早是基于 Mr.doob 渲染 Flash 2D 图形的工作，所以使用 HTML5 Canvas 渲染器是一个自然的转变。事实上，Canvas 渲染器还比 WebGL 渲染器更早实现。

Three.js 中有一大堆基于 Canvas 的例子。但不幸的是，它们大部分都不吸引人。不过有几个值得在这里提一下。在 Three.js 项目的源文件中，打开 examples/canvas_geometry_earth.html，如图 7-7 所示。你会看到一个带纹理映射的旋转地球。它并不比 WebGL 的版本漂亮，但还是很不错的。你最明显察觉到的应该是球体不是完全地契合；这就是说，没有太多的矩形来绘制它。你可以在旋转的时候看到三角形边缘。尽管并不像高尔夫球那么夸张，但它还是比较粗糙。原因就是三角形排序。你必须控制三角形的数量，因为深度排序三角形是一个最少 $O(N\log N)$ 的操作，所以三角形更多意味着排序更慢。



图 7-7：使用 Three.js 的 Canvas 渲染器绘制的纹理映射地球

Canvas 渲染很适合用来做简单的 3D 全景图。在这里我们将讨论渲染 12 个三角形（立方体内部有六个面，每个面需要渲染两个三角形），所以三角形数量并不是个问题。打开 Three.js 例子中的 examples/canvas_geometry_panoramas.html 来查看 3D 全景效果，如图 7-8 所

示。使用鼠标来旋转场景，导航起来很平滑，而且全景效果看起来不错。



图 7-8：使用 Three.js 的 Canvas 渲染器绘制的全景图

Three.js 的 Canvas 渲染同样擅长绘制大量简单的 2D 多边形，如扁平的 2D 多边形，摆放在 3D 空间中。这是一个创建类似粒子动画这样花哨页面效果的好方法。图 7-9 展示了一个例子（源文件 examples/canvas_particles_random.html），当鼠标在屏幕中移动的时候，有 1000 个随机的粒子在跟着动。这些形状是扁平的，但它们在 3D 空间中移动。想象一下，如果使用 CSS 3D 变换来实现这个效果，1000 个独立的元素会让浏览器不堪重负。但使用 Canvas 就很轻松了，而基于 Three.js 就更容易创建了。

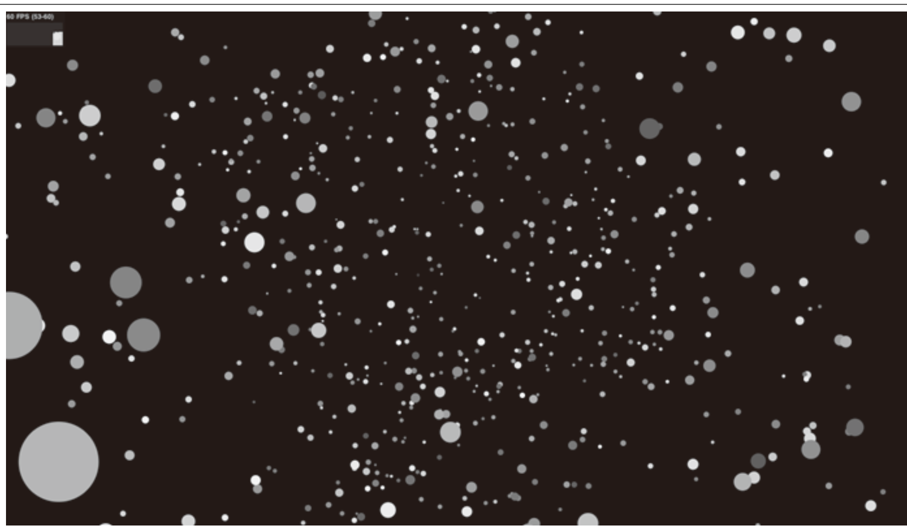


图 7-9：使用 Three.js Canvas 渲染器渲染的 1000 个粒子动画

1. 如何使用Three.js的Canvas渲染器

让 Three.js 使用 Canvas 很简单，只需要创建一个不同类型的渲染对象。但还有其他微妙的操作。让我们看一个简单的例子。在其中，我们会探索一些 Three.js Canvas 渲染器和 WebGL 渲染器在展示效果和性能上的不同。最后，我们会在实际例子中测试。在这之前演示的例子都不够真实，更像是技术演示。让我们尝试通过多边形模型及纹理来创建一些具体的、类似游戏的图形。

图 7-10 展示了使用 Three.js Canvas 渲染器来制作游戏的效果。它是一个用来评估视觉质量和帧率的简单查看器。在浏览器中打开文件 Chapter 7/threejscanvasmodel.html，你会看到在一个简单星空背景上缓慢旋转的三艘太空船。你可以使用鼠标来旋转场景，使用滚轮来放大和缩小。你可以通过点击 Animate 复选框来开始和停止动画。这个例子还允许你切换使用 Canvas 及 WebGL 来进行比较。让我们先来研究 Canvas 的版本。

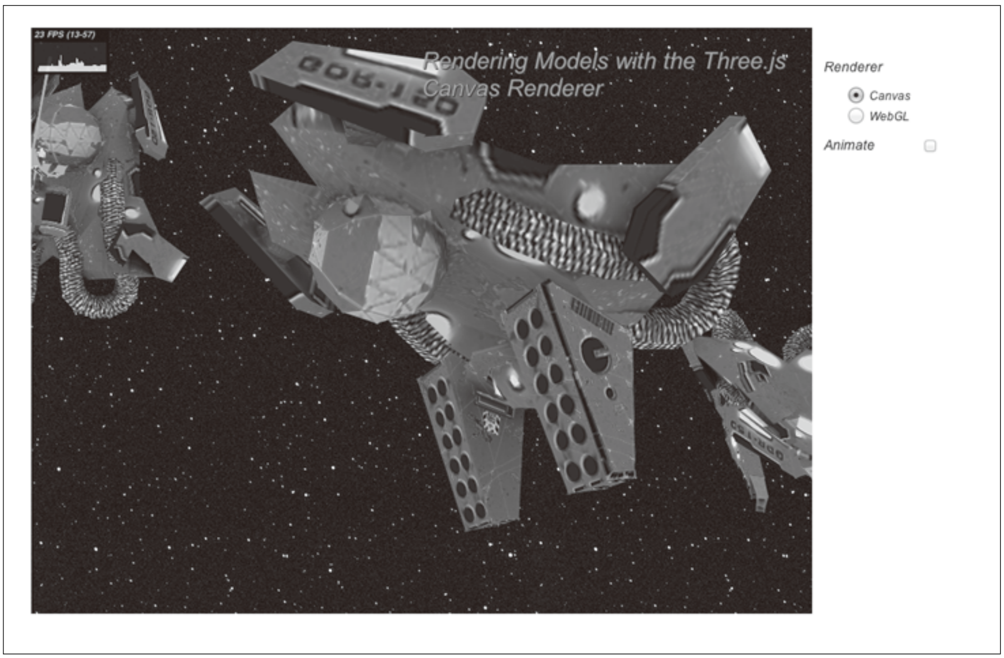


图 7-10：使用 Three.js 的 Canvas 渲染器来渲染模型；太空船模型来自 Turbosquid（http://www.turbosquid.com/FullPreview/Index.cfm/ID/499274），作者是 gentlemenk（http://www.turbosquid.com/Search/Artists/gentlemenk）

注意左上角的帧率，它处于 20~23fps。如果你停止动画并放大场景，让一艘太空船从视野中消失，你会发现帧率忽然增加到了 30fps 左右。你可以再尝试让视野中只剩下一艘太空船，你会发现帧率到了 50fps 左右，甚至 60fps。这清晰演示了我们之前提到的三角形排序问题。因为 Three.js 的 Canvas 渲染器没有深度缓存，所以它需要进行三角形排序。越多的三角形意味着越慢的排序速度。但我们放大到只看见一艘船的时候，Three.js 很聪明地将其他物体忽略（剔除）掉了，所以它们不需要进行三角形排序。这些太空船模型都很简单，每个由大约 1200 个三角形组成。这对于现代游戏来说并不多，所以这个例子说明了

我们使用 Canvas 渲染时需要节约多边形的使用到何种程度。现在来看看它的材质。太空船被照亮了，而且场景还有一个定向光源，它应该照亮太空船中的各种部件；然而，我们并没看到那个效果。Three.js 可以应用一些光源，但效果都很基础，我们并没有看到物体表面上光滑的着色效果。你可以尝试一下 Three.js 中的其他 Canvas 效果，来看看材质和光源能做得怎样。

2. 比较Canvas渲染器和WebGL渲染器

现在我们切换渲染器来进行比较，点击 WebGL 的单选框按钮来使用 WebGL 来渲染场景。如图 7-11 所示。视觉对比非常明显。在 WebGL 版本中，纹理看起来很平滑，尤其是物体在远处的时候，而在 Canvas 版本中颗粒状明显。边缘的反锯齿也比 Canvas 版本更加平滑。最引人注目的差别是光照，我们可以清楚地看到定向光源带来的高亮效果，而这在 Canvas 版本中是没有的。而在性能方面，看看帧率你会发现即便所有太空船都在场景中，它依然稳定保持在 60 fps。这并不令人吃惊，因为 Three.js 只需做很少的工作。在场景中只有少数几个物体，多边形数量不多且纹理简单。几乎所有的计算都是在硬件中完成的（也就是说，在 Three.js WebGL 渲染器内的 GLSL 着色器代码中）。

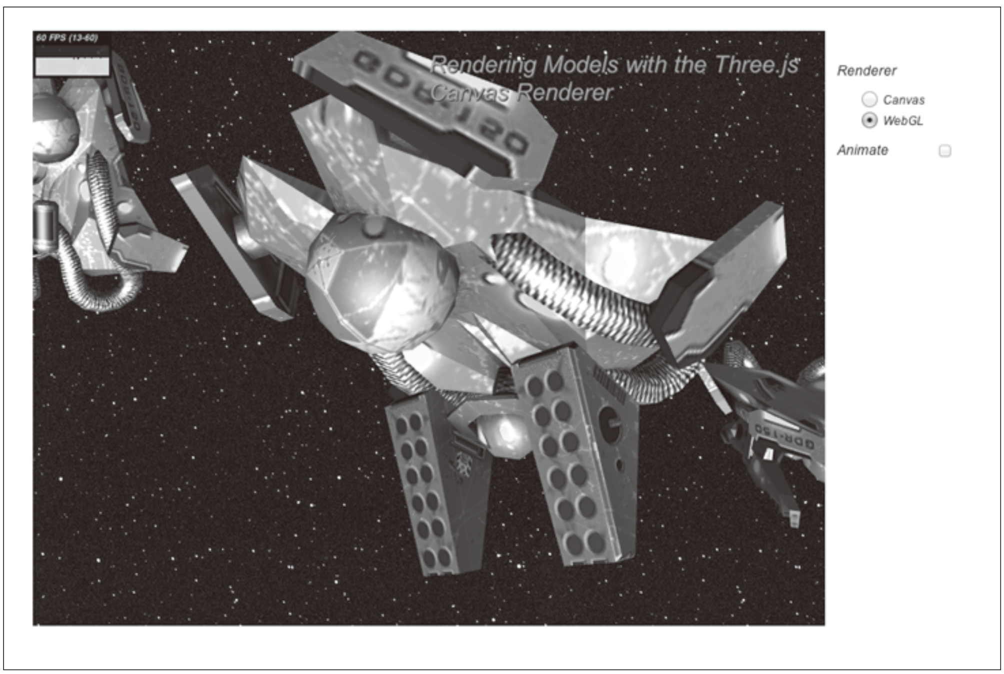


图 7-11：使用 Three.js WebGL 渲染器渲染的太空船

尽管这些或许看起来描绘了一幅使用 Canvas 做 3D 游戏的沮丧画面。我们在简单的场景下就达到了帧率的极限，使得我们不得不对视觉质量妥协。但这是悲观主义的看法。从另一个角度上说，我们能够只使用 JavaScript 就在页面中创建了上千个带纹理的三角形，这已经很令人骄傲了。如果我们开发简单的游戏，并且可以创建一种在局限条件（例如低多边形和预渲染光）下的艺术风格，同样可以做出令人惊叹的事情。

只需要几行代码就能让 Three.js 使用 Canvas 渲染器。下面这个例子的源码可以在 Chapter 7/threejscanvasmodel.html 中找到。例 7-3 给出了创建渲染器的代码。注意粗体标注的代码行，我们创建了一个 `THREE.CanvasRenderer` 类型的对象，而不是 WebGL 渲染器。

例 7-3：创建 Three.js Canvas 渲染器

```
function createRenderer(container, useCanvas)
{
    if (useCanvas) {
        renderer = new THREE.CanvasRenderer( {  } );
    }
    else {
        renderer = new THREE.WebGLRenderer( { antialias: true } );
    }

    container.appendChild( renderer.domElement );

    // 设置视口大小
    renderer.setSize(container.offsetWidth, container.offsetHeight);
}
```

一旦 Canvas 渲染器创建好后，我们就能以和 WebGL 相同的方式来渲染：

```
        // 渲染场景
        renderer.render( scene.scene, scene.camera );
```

对于简单的使用，这确实是我们唯一需要更改的一行代码。但在这个例子中，我们还需要做另外一个事情。Three.js 让我们可以通过“过度”绘制三角形的方法来实现简单的边缘反锯齿，它的做法是在绘制的时候都多绘制一个像素，用来隐藏三角形间的缝合处。但不幸的是，我们不能在渲染器中设置一个 `antialias` 标记（在 WebGL 中我们可以这么做），而需要在每一个材质上设置。这需要在模型加载后遍历它的材质，然后设置 `overdraw` 属性为 `true`。查看例 7-4。我们在每个模型加载完成的时候设置了一个回调函数。这个回调函数会调用模型中的 `traverse()` 方法进行遍历，然后访问每个场景图中的后代。我们的辅助函数 `processNodes()` 先测试对象是否是一个网格，如果是的话，就设置网格上的 `overdraw` 属性。这个额外的工作有点麻烦，但总的来说需要做的事情还是不多的。这两处修改就是使用 Canvas 和 WebGL 渲染的唯一区别。

例 7-4：遍历场景中的材质

```
function processNodes(n)
{
    if (n instanceof THREE.Mesh)
    {
        n.material.overdraw = true;
    }
}

function handleSceneLoaded(data, parent)
{
    // 将网格添加到分组中
    parent.add( data.scene );
    data.scene.traverse(function(n) { processNodes(n) });
}
```

7.4 小结

本章详细说明了如何使用 2D Canvas API 来实现 3D 的软件渲染。所有 HTML5 浏览器都支持 Canvas API。在简单介绍了 Canvas API 的绘制功能后，我们研究了软件渲染中固有的一些问题，包括变换、着色和深度排序。我们学习了基于 Canvas 的 3D 库，特别是如何使用 Three.js 的 Canvas 渲染器来替代 WebGL。这里面有许多折衷，尤其是性能和视觉保真方面。Canvas 提供了一个可行的 WebLG 替代方案，尤其是在简单、有限的使用场景下，可以使用它来作为不支持 WebGL 平台上的降级方案。

第二部分

应用开发技术

第8章

3D内容制作流程

在 Web 发展的早期，如果你知道如何写标签，你就是 Web 内容的创建者。当时没有 Dreamweaver 这类所见即所得的编辑器，也没有 PhotoShop 这类切图工具。在当时，Web 内容创建的绝大部分工作都由程序员来完成——而那个时期的 Web 也恰恰反映出了这个特色。后来，软件开发商们发布了创建 Web 内容的专业编辑工具。当这些工具流行起来之后，艺术家和设计师们也逐渐承担起创建内容的职责，互联网的体验终于提升到了消费级的水平。

WebGL 开发正在经历和早年 Web 类似的演变。该技术最早出现的几年中，内容是由程序员利用文本编辑器手工编写创建的，或者使用能够找到的转换器将其他 3D 格式的内容转换为 WebGL 所支持的格式。如果找不到现成的转换工具，那么为了完成项目，你可能需要自行编写一个。

幸运的是这种情况正在快速转变。Three.js 和其他 WebGL 库在导入专业工具创建的 3D 内容这方面表现得越来越强大。业界还齐心协力创建了专供 Web 使用的 3D 文件格式标准。虽然从整体来说模式依旧比较原始，但对比几年前，WebGL 内容的开发可以说已经从粗放的“石器时代”完成了向至少有可用工具的“青铜时代”的跨越。

本章涵盖了 Web 3D 内容的开发流程。首先，我们会通览内容创建的过程。如果你对 3D 创作不熟悉，那这部分内容对你来说会非常有用。其次我们会大体了解现今 WebGL 项目中使用的流行建模和动画工具，并深入研究最适合用于 Web 开发的 3D 文件格式细节。最后，我们会学习使用 Three.js 的通用工具将这些文件加载进来，为后续章节的项目做准备。

8.1 3D内容创建过程

3D 内容的创建涉及一系列高度专业化的学科。3D 工作者的整个职业生涯需要广泛的培训

以及对复杂创作工具和流程的深入理解。一名 3D 艺术家通常需要负责所有的事情，包括建模、纹理映射以及动画。但有时候，尤其是在比较大的项目中，人们会进行分工。

3D 内容的创建在很多方面与使用 Photoshop 或 Illustrator 来创建 2D 艺术作品相类似。但 3D 创作和 2D 艺术作品的创建有一些本质上的不同。即便你是一名技术人员，当你打算开发一个 3D 项目的时候，最好还是对被导入到项目中的内容的产出过程有一定的了解。怀着这个目的，让我们来看看 3D 内容创建过程中的基本步骤。

8.1.1 建模

在 3D 模型的创作中，通常先由艺术家绘制一个草图。然后使用建模软件包将草图转换为 3D 数字表现。模型通常用 3D 多边形网格来表示，首先通过线框图来绘制，然后使用材质来着色。这个过程被称为 3D 建模（3D modeling）。专门从事这项工作的人被称为建模师（modeler）。图 8-1 描绘了一个由 Autodesk 3ds Max 创建的简单茶壶模型。我们可以从四个不同的视图观察该模型：顶部、左边、前面和透视图。

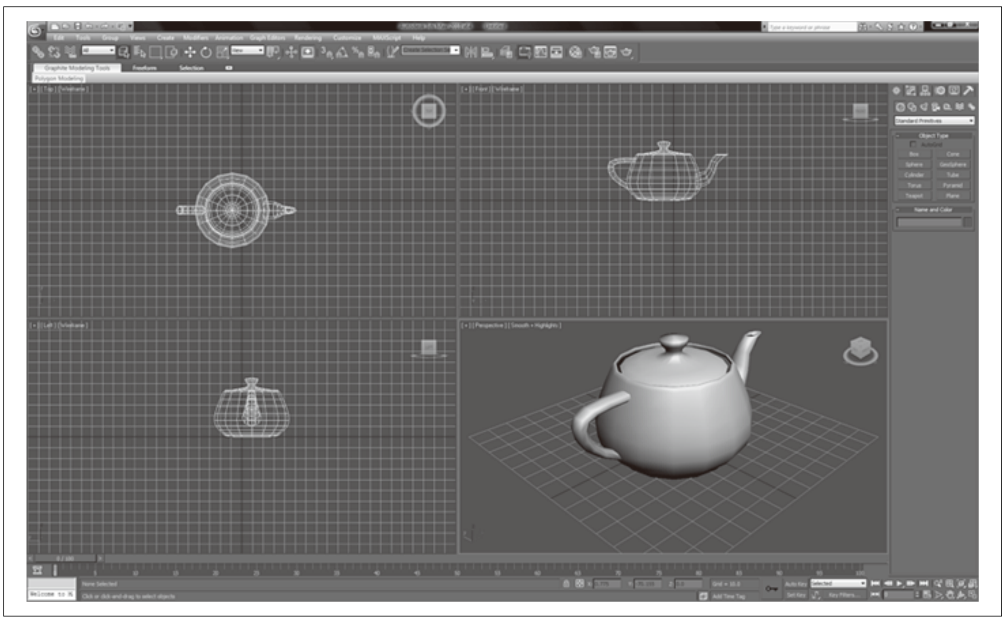


图 8-1：3ds Max 中的 3D 建模，有顶部、左边、前面和透视图四个视图；图片版权为 Autodesk 所有，来自维基百科词条（http://en.wikipedia.org/wiki/File:3dsmax_2010_800px.png）

8.1.2 纹理映射

纹理映射（texture mapping，也称为纹理贴图），也被称为 UV 映射（UV mapping，也称为 UV 贴图），是将创建好的 2D 素材展开附着到 3D 物体表面上的过程。建模师通常会自己进行纹理映射，但在比较大的项目中，贴图这项职责可能会被划分出来，由专门的贴图师（texture artist）来负责。纹理映射一般由内建在建模软件包中的可视化工具来辅助完成。

这类工具让贴图师可以通过可视化的方式将 2D 纹理图和网格中的顶点关联起来并预览效果。图 8-2 描绘了纹理映射。图的左边是纹理；结合视图在右下方，可以看到图片数据覆盖在顶点的位置上；预览效果在右上方。注意左边的纹理图布局有点违反直觉，只显示了半边脸。这是因为对于这个纹理映射来说，左右脸是一样的。这个策略允许贴图师在较小的空间里放更多的数据，并且使用图片的其他部分来增加更多细节。

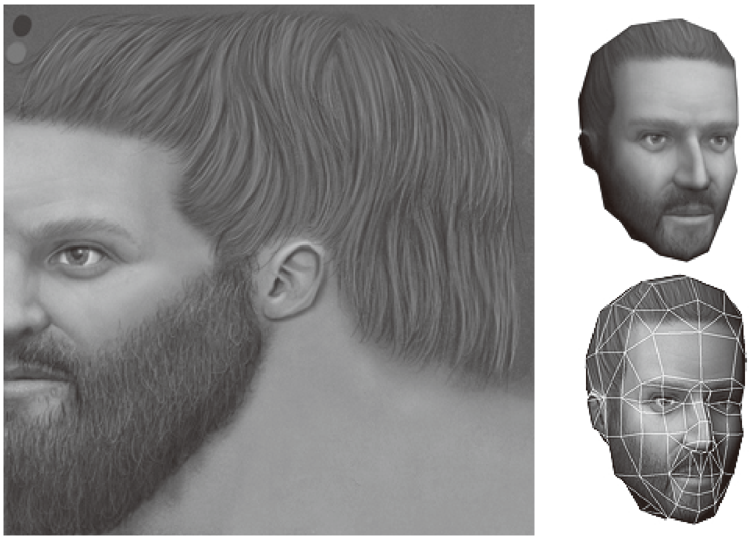


图 8-2：纹理映射——一个 2D 图像包裹并映射到一个 3D 物体的表面；图片由 Simon Wottge 提供（http://www.simonwottge.com/?cat=13）

8.1.3 动画

创建 3D 动画的过程可以很简单也可以极其复杂，这取决于任务本身。关键帧动画往往比较简单，至少从理论上来说是这样的。界面可能会难以使用且视觉上杂乱。一个关键帧编辑器，就像图 8-3 Autodesk Maya 展示的那样，包含一套时间轴控制器（在 Maya 窗口底部使用矩形高亮部分），它允许*动画师*在视图中移动或改变物体，然后在时间轴上点击来创建关键帧。关键帧可以用来改变变换、旋转、缩放，甚至是光线及材质属性。当动画师需要在关键帧中定义多个属性的变换时，他需要在动画时间轴上增加额外的轨道。动画师在界面中通过堆叠的方式排列这些轨道，这会导致视觉上的混乱。

对有蒙皮的角色进行动画会牵涉更多的工作。在可以给角色设置动画前，需要先创建一系列的骨骼，也被称为*骨架*（rig）。骨架决定了蒙皮跟随骨骼移动时的各种特性。创建骨架的过程是需要专业技巧的，角色的动画及骨架经常由不同的动画师完成。

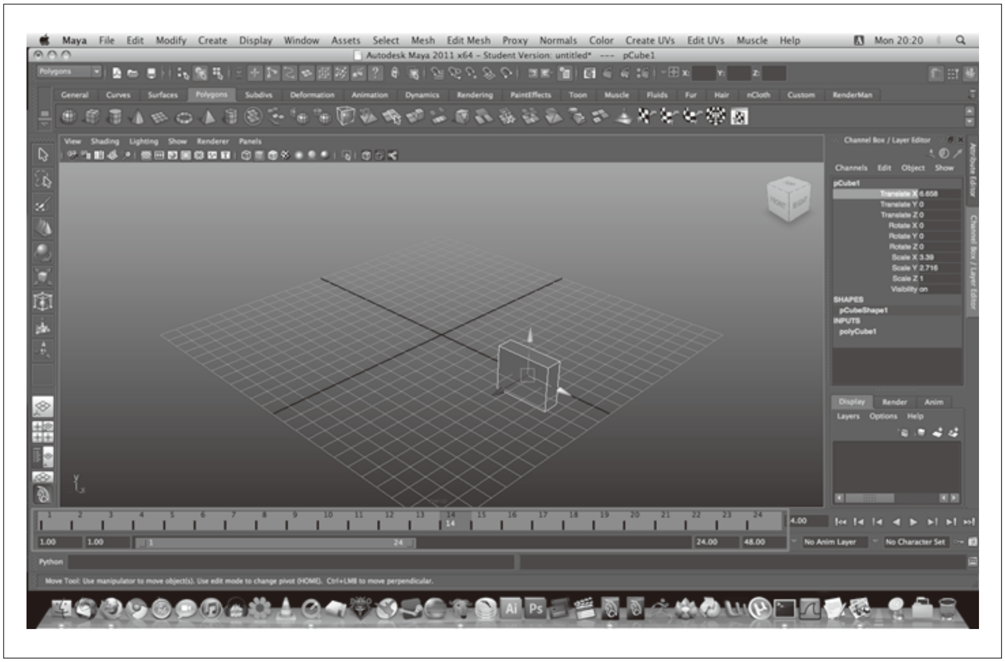


图 8-3：Maya 的动画时间轴工具，具有对变换、旋转、缩放及其他属性变换的关键帧的控制；图片来自 UCBUGG Open Course Ware（http://ucbugg.github.io/learn.ucbugg/introduction-to-maya/）

8.1.4 技术美工

我们可能没想到编程也是内容创建活动的一部分，但在 3D 开发领域，它通常是。复杂的特殊效果，比如某些着色器和后期技术处理，需要有经验的程序员来完成。在游戏和动画公司里，这些工作被交给*技术美工*（technical artist）或*技术总监*（technical director）来做。对于这些职位并没有一个正式的定义，在这两个职位间也没有严格的区别。但根据名字来判断，技术总监通常是更资深和有经验的人。技术总监需要写脚本、给角色创建骨架、写转换程序来将一种格式转成另一种格式、实现特殊的效果、开发着色器，换句话来说就是所有那些对于视觉设计师来说太过于技术化的工作。这是高度有价值的技能，对于许多产品来说，一个好的技术总监价值连城。

因为技术总监主要靠写代码为生，他们所使用的工具通常是文本编辑器。但是，现在有些有趣的可视化开发工具供开发着色器和特殊效果使用。其中之一是 ShaderFusion，一个最近在 Unity 游戏引擎中发布的可视化工具。ShaderFusion 允许开发者通过定义从一个物体的输出（比如时间或位置）到另一个物体的输入（例如颜色或折射）之间的数据流来开发着色器。它的界面如图 8-4 所示。

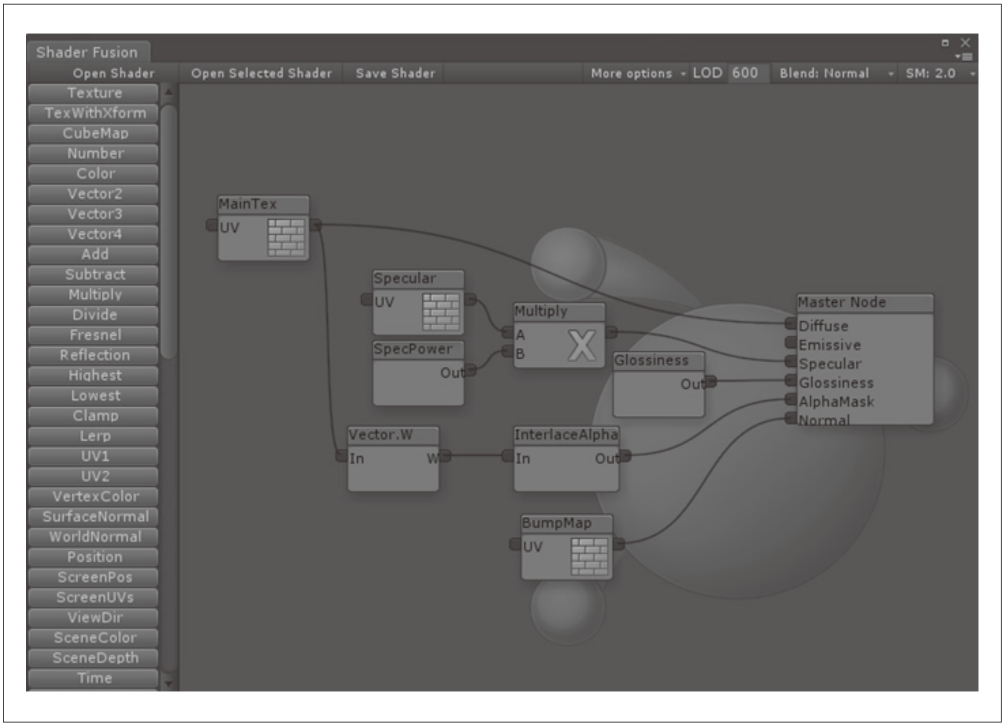


图 8-4：ShaderFusion（http://www.shaderfusionblog.com/），一个 Unity3D 引擎中的可视化着色器编辑器

8.2　3D建模和动画工具

本节我们介绍 3D 设计师用来创建内容的流行工具。既有传统的桌面软件产品，能够满足不同层次的用户和技能；也有新出现的看起来有前景的基于 HTML5 的云服务工具，其中有些服务是免费的，而有些通常需要每月付费。最后，设计师还可以从一些在线网站下载已存在的模型和动画，来借用其他同行的工作成果。

8.2.1　传统3D软件

大部分时候，3D 内容创建是在数字内容创建（Digital Content Creation，DCC）工具中完成的。3D DCC 工具最早出现在电影制作和工程领域，现在被广泛用于建筑设计、游戏开发、静态艺术渲染及更多领域。可以把这些软件类比为 3D 领域的 Adobe Photoshop。它们在 Web 产品流程中占据了类似的位置：作为最初素材的来源，并被进一步转换、优化和集成到页面中。

3D DCC 工具通常是操作系统的本地应用，或者叫“盒装软件”（当然现在基本上是从开发者的网站上下载的）。3D DCC 工具需要专业技能，并且有复杂的用户界面和陡峭的学习曲线。好消息是越来越多的数字设计师在学校及早期职业培训中就开始学习这些工具了。像

常驻的 Photoshop 专家一样，随着你的 3D 项目不断发展，有经验的 3D 设计师会成为 Web 团队中的一员。

3D DCC 工具的价格差别很大，从完全免费的 Blender，到每个许可费高达几千美元的产品，如 Autodesk 的 3ds Max 和 Maya。这些工具通常会包含一套通用的功能，包括建模、贴图、动画；然而，有些产品专攻这些领域中的某一项。大部分 3D DCC 工具内置了导入器和导出器，能够导入和导出我们待会将会介绍的标准格式。它们还有某些形式的可拓展性，比如原生（基于 C++ 的）SDK，或者提供高级语言脚本来编写插件，以增强界面，提供自定义渲染，导出新的文件格式等。

接下来对广泛使用的建模及动画工具进行介绍，你也许会在开发 WebGL 项目的时候用到它们。本章后面，我们会研究如何将它们中的一些集成到 WebGL 内容创建流程中。

1. Autodesk 3ds Max、Maya以及MotionBuilder

坐落在加州圣拉斐尔的 Autodesk 公司，是市场上三个最流行的建模及动画产品的制造商，它们分别是：3ds Max、Maya 和 MotionBuilder。MotionBuilder 主要用于角色动画，3ds Max 和 Maya 则是全能的 3D 工具。3ds Max 和 Maya 在功能覆盖方面很类似，所以对于新人来说很难选择用哪个。现有用户主要根据自己的喜好、工作流程的偏好等因素来进行选择。它们之间有一个很大的区别就是 Maya 可以运行在 Windows 和 Mac 上，而 3ds Max 只支持 Windows。这三个产品都能输出 Autodesk 公司的通用格式：FBX。

> 为什么 Autodesk 会有如此多相似的产品？大概十年前，这个公司有点购物狂，买下了很多竞争产品，从 Alias Systems 公司买下了 Maya，从 Kaydara 买下了 MotionBuilder。MotionBuilder 主要用在角色动画上，而其他两个产品则有相同的功能。在 Tom's Hardware 网站上有一篇内容充实的文章对 3ds Max 和 Maya 进行了比较（http://www.tomshardware.com/forum/247220-49-maya）。

Autodesk 的工具有着复杂的用户界面，有大量控件、视图、属性编辑器及弹出窗口。它们是完整 3D 开发中“工作站”类型的产品。界面通常最开始是像图 8-1 的 3ds Max 截屏那样的四个视图，也可以收缩到图 8-5 May 截图那样的单一场景视图。这些产品的通用功能包括材质编辑器、创建基本几何体的工具栏、绘制和编辑网格的工具、动画时间轴工具、渲染插件、着色器编辑器，等等。

Autodesk 的工具是为专业人士定价的：每个产品大概 3000~4000 美元。公司同时还提供了基于订阅的价格，以及针对学生和学习者的版本。



图 8-5：Autodesk Maya，一个完整的 3D 建模及动画套件；图片版权为 Autodesk 所有，来自维基百科词条（https://en.wikipedia.org/wiki/File:Autodesk_Maya_2013_SP2_Extension_x64_on_Win8.png）

2. Blender

Blender（http://www.blender.org/）是一个免费、开源且跨平台的 3D 创作工具。它可以运行在主流的操作系统上，以 GNU GPL（General Public License，通用公共许可证）协议的方式授权。Blender 是由荷兰软件开发者 Ton Roosendaal 创建的，它由 Blender 基金会维护，这是荷兰的一家非营利性组织。Blender 非常流行，基金会估计有两百万用户。它的用户有设计师和工程师，从业余爱好者 / 学生到专业级别都有。

像 3ds Max 及 Maya 那样，Blender 的用户界面复杂，有多个视图、工具栏及控件，并伴随着陡峭的学习曲线。所以尽管是免费的，它并不适合意志薄弱的人。尽管如此，Blender 对 Web 开发者而言是一个有吸引力的选择，有以下几个原因。

- 它是免费。
- 它是开源的。
- 它可以使用 Python 进行扩展。
- 它支持导入和导出多种文件格式，包括 3ds Max、OBJ、COLLADA 和 FBX。Three.js 开发者也开发了一个从 Blender 导出到 Three.js JSON 格式的工具（本章后面会介绍）。

3. Trimble SketchUp

一个中级 3D DCC 工具是 SketchUp（官方叫法是 Trimble SketchUp），它是一个易于使用的 3D 建模工具，可用于建筑架构、工程，有时也用于游戏开发。

SketchUp 的历史很有趣。它最开始由 @Last 软件公司于 1999 年开发，后来得到了 Google Earth 团队的注意，他们基于 @Last 的工作开发了一个系统插件。而后，Google 在 2006

年收购了 @Last。多年来，SketchUP 被推广为 Google Earth 中用户创建 3D 建筑及地标的工具。SketchUp 还有一个姊妹网站——3D Warehouse，一个由业余创作者上传和分享 3D 模型的在线仓库。2012 年，Google 决定退出经营用户生成 3D 内容的相关业务，然后将 SketchUp 卖给了 Trimble Navigation，一家位于加州的 GPS 系统制造商。Trimble 继续发布 SketchUp 和维护 3D Warehouse，但它不再用来为 Google Earth 生成内容。

SketchUp 运行在所有平台上。它的价格很合理，专业版大概 500 美元。还有一个针对业余用户的完全免费版。SketchUp 因易于使用而被熟知，它使用了基于线条绘制的方法来建模，这对架构师和工程师很有用。SketchUp 具有优秀的 COLLADA 导出能力（请查看 8.3 节中的“COLLADA：数字资源交换格式”），所以它可能是 WebGL 开发一个不错的选择。SketchUp 可以从它的官方网站（http://www.sketchup.com/）上下载。

4. Poser

Smith Micro 的 Poser 是一个中级角色动画工具。像 SketchUp 那样，它的价格足够吸引人，而且目标用户是业余内容创作者。Poser 有着直观的用户界面来对角色进行姿势摆放和动画。它有一个庞大的库，包含已经建好模型、骨架、带完整贴图的人物和动物角色，以及一系列背景场景、道具、载体、相机和光照设置。Poser 可以用来创建相片般真实的静态渲染和实时动画。Poser 的用户界面请查看图 8-6。

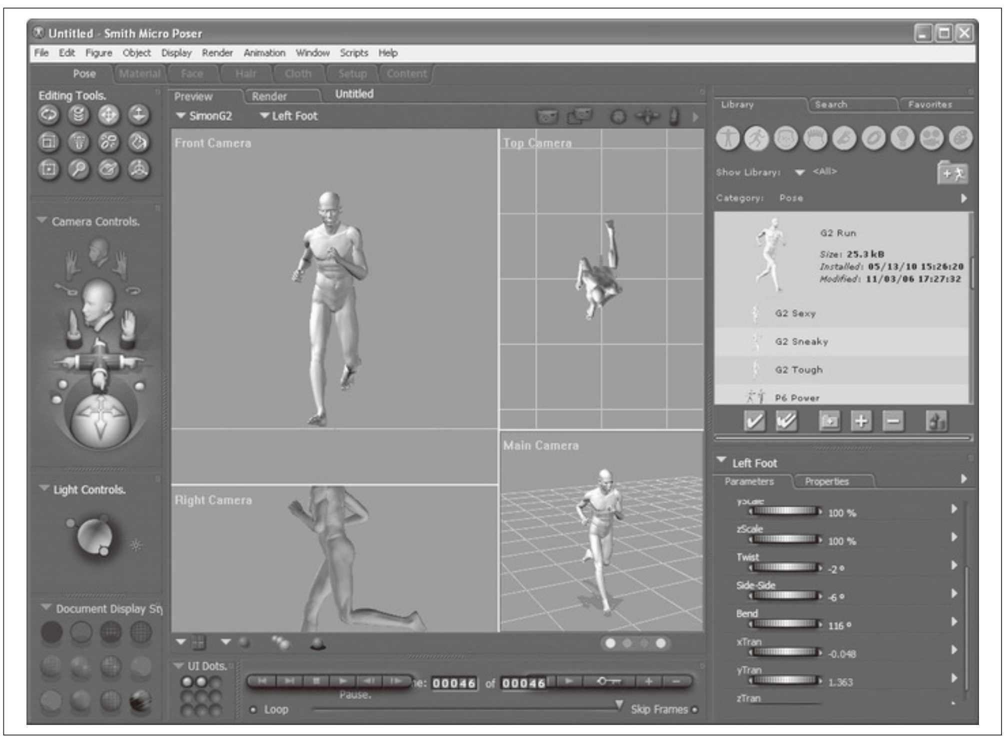


图 8-6：Poser 的用户界面（http://poser.smithmicro.com/）；图片来自 Smith Micro Software 公司

在所有讨论过的软件里，Poser 值得一提的是它的开发团队从 COLLADA 文件格式发展

初期，就很积极地参与它的创建，同时这个团队还积极和 Khronos 组织合作开发新的标准：glTF，我们将在本章后面进行讨论。Poser 团队对标准格式很赞同，把它当成一种能让 3D 内容得到广泛支持的方式，尤其是在 Web 环境中。Smith Micro 的高级工程总监 Uli Klumpp 就在 WebGL 中使用 Poser 说了以下这段话：

> 支持 WebGL 的应用和支持其他格式的应用并没有什么不同，都需要描绘人体形式（或明确的非人体形式）。Web 设计师从 20 世纪 90 年代就开始使用 Poser 来证明这一点。他们终于获得了一个无处不在的 3D 内容展示场所，而 Poser 巨大的内容库已经准备好了。

8.2.2 基于浏览器的集成环境

HMTL5 及廉价云计算的出现为新的 DCC 工具——基于浏览器的集成 3D 开发环境——的诞生创造了条件。建模和动画依然在之前提到的本地工具中创建，但场景布局、交互编程和 Web 发布可以在浏览器界面中进行。

基于浏览器的环境提供了与它们的桌面对手不一样的独特特性。首先，显然，它不需要下载；其次，它们本身就是用 WebGL 开发的，所以可以提供与部署后效果一样的所见即所得环境。基于浏览器的工具通常会有吸引人的价格，并使用增值收费的模式，允许开始使用的时候免费，只有当开发者进行商业化的时候才收费，比如开发一个团队项目，或者使用的文件存储超过一定上限。这是一个全新且不断变革的领域，所以开发者可以期待在接下来的几年里一个 Web 风格的持续收费模式及价格。

1. Verold

Verold Studio 是一个轻量级 3D 交互内容发布平台，由位于多伦多的 Verold 公司开发。它不需要依赖插件，可以使用简单的 JavaScript API 进行扩展，因此业余爱好者、学生、老师、可视化通信专业人员以及 Web 市场人员可以简单地将 3D 动画内容集成到他们的 Web 中。

典型的 Verold 工作流程需要一个 CG 设计师上传相关的资料（3D 模型、动画、纹理）到一个 Verold 项目中。协同工具可以用来提供资料的反馈；编辑工具可以用来建立材质及着色器，并布置到场景 / 关卡中。当团队对资料设置结果满意后，Web 设计师可以导出样例代码到目标页面中。这个工作流程适合很多场景，无论是与开发者和 CG 设计师坐在一起，还是远程协作，抑或是有些场景下的资料是买来的而不是自己开发的都行。Verold Studio 的用户界面如图 8-7 所示。注意它基于浏览器的简洁设计，这与传统复杂的 DCC 工具形成鲜明对比。

Verold 可以实时协作编辑、线上发布以及共享内容，开启了 3D 项目开发的新方式。来看看创始人及 CTO Ross McKegney 的说法：

> 一个使用 Verold Studio 的好案例是 Swappz Interactive 公司。Swappz 为“忍者神龟：变种时代”“蓝精灵”“恐龙战队”等品牌创造玩具。这些玩具有点特别，它们可以被相关的移动游戏“扫描”进去。Swappz 在开发过程中使用了 Verold，用来保证跨境的角色设计师及当地的动画师之间能够相互反馈、为总公司显示进度、从 Nickelodeon 公司获得使用资源的许可，以及最终当游戏准备好发布的时候，将游戏资源用在游戏推广网站上。



图 8-7：Verold Studio（http://www.verold.com）

2. Sketchfab

另一类在线 3D 工具提供了上传并分享的服务。3D 设计师可以上传几种格式的 3D 模型，使用 WebGL 来在线预览和分享。这其中开发最完善的是 Sketchfab（http://sketchfab.com/），它由位于巴黎的团队开发，团队创始人有 Cédric Pinson、Alban Denoyel 和 Pierre-Antoine Passet。Sketchfab 是一个实时发布和共享交互式 3D 模型的在线服务，它不需要插件。只需点击几下，设计师就可以上传一个 3D 模型到网站上，获得一段 HTML 代码，它可以嵌入到其他地方，展现的时候使用 Sketchfab 提供的服务来进行渲染。

Sketchfab 支持几种原生的 3D 格式，以及大部分着色器，包括法向量贴图、镜面反射、凹凸贴图、漫反射等。Sketchfab 同样提供了一个材质编辑器，让设计师能够在浏览器中实时调整着色器和渲染效果。它还开发了主流本地 DCC 工具的导出器，模型可以直接从编辑工具（比如 Maya）中导出，这样会更方便些。Sketchfab 的首页如图 8-8 所示。页面中主要区域的图像实际上是 Sketchfab 库中的一个模型，使用了 WebGL 渲染。

3. SculptGL

由于 3D 渲染和用户界面的限制，几年前开发在浏览器中进行 3D 建模的工具是不可想象的。但现在有了 HTML5 和 WebGL，这个想法就再不荒诞了。Stephane Ginier 开发了 SculptGL，一个基于 Web 的实体建模工具，其界面简单易用，可用来创建简单的、雕塑风格的模型。SculptGL 是免费且开源的，可以在 GitHub 上找到，地址是 https://github.com/stephomi/sculptgl。SculptGL 可以导出多种格式，并可直接发布到 Verold 和 Sketchfab 上。SculptGL 的界面如图 8-9 所示。

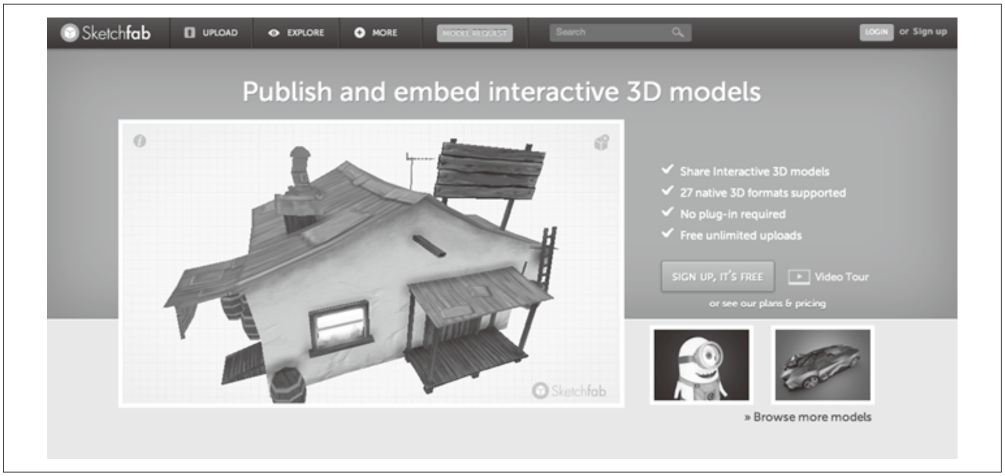


图 8-8：Sketchfab 网站（http://sketchfab.com/）允许内容创建者上传并分享可以实时浏览的 3D 模型



图 8-9：SculptGL（http://stephaneginier.com/sculptgl/），一个基于浏览器的开源 3D 模型工具

4. Shadertoy

类似 JSFiddle（http://jsfiddle.net/）那样的 Web 上的“沙箱工具”越来越流行。JSFiddle 允许开发者在浏览器中测试代码，并实时预览效果。鉴于此，必然也会有人开发用于 WebGL 的类似沙箱工具。Shadertoy（https://www.shadertoy.com/)）是一个基于浏览器的代码工具，它可以编写和测试 GLSL 着色器。它包括沙箱和在线社区。一旦一个着色器编写完成并测试通过，它就可以被提交到 Shadertoy 网站上供人发现并使用。这是一个学习 GLSL 着色器编写的好方式，你可以模仿其他人的工作。当一个着色器开发好后，你可

以通过 Shadertoy 网站来分享，或者将 GLSL 代码复制到你的应用源码中。图 8-10 展示了 Shadertoy 的界面，它包括一个实时预览框、一个代码编辑框、一个选择着色器来源的交互式图标。

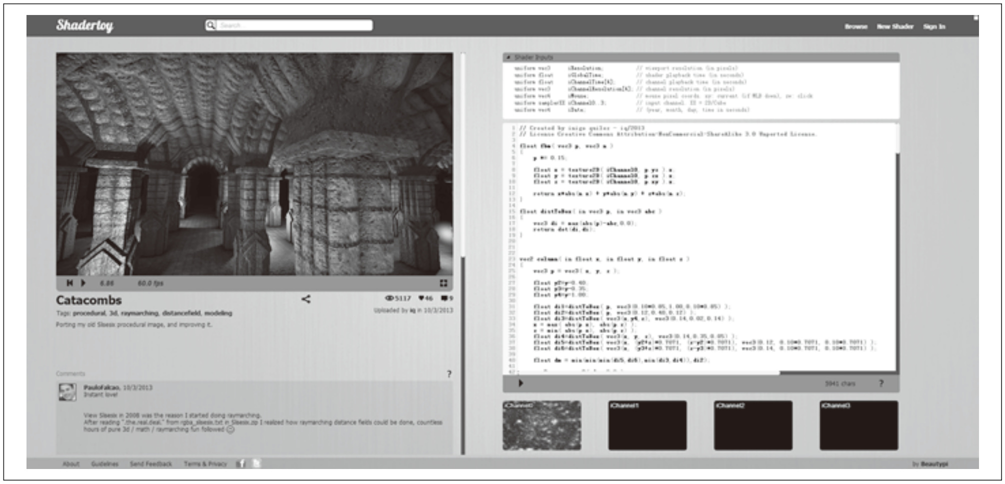


图 8-10：Catacombs——一个使用过程纹理的 Shadertoy 实验（https://www.shadertoy.com/view/lsf3zr）

8.2.3　3D内容仓库和现成素材

并不是所有人都有 3D 建模的才能，并且由于项目预算和时间的限制，我们无法一下子招到合适的人。好在还有不错的 3D 内容在线资源，类似于 3D 版本的“剪贴图集”。模型和内容包的价格从免费到几百甚至几千美元不等，质量也千差万别。有些创作者允许无限制的使用，而有些则有所限制。注意好好查看授权许可，尤其是开发在线传播的 Web 应用。

为了给这本书创建 3D 内容，我们使用了许多不同在线资源的模型，列举如下。

- The Trimble 3D Warehouse（http://sketchup.google.com/3dwarehouse/）

  3D Warehouse 最早由 Google 开发，用于支持业余爱好者上传世界建筑和地标的 SketchUp 模型，并在 Google Earth 中标注它的位置。Trimble 收购 SketchUp 后，这个服务不再用于 Google Earth，但依然是一个优秀的已渲染建筑和其他 3D 物体的重要来源。

- Turbosquid（http://www.turbosquid.com/）

  Turbosquid 成立于 2000 年，是包含成千上万个模型的一流网站，这些模型用在动画、游戏及建筑架构中。许多模型的多边形数量为少或中，对于实时渲染及 Web 很适用。

- Renderosity（http://renderosity.com/）

  Renderosity 成立于 1998 年，是一个多样化的社区，有许多 2D 和 3D 的创意专家。这个网站有许多模型和纹理。它主要专注于用于预渲染静态图像的高多边形模型，而不是用于实时渲染的低多边形模型。

- 3DRT.com

  3DRT 是一个严谨的在线商店，它有用于实时游戏及 Web 的高质量 3D 素材。这个网站的组织形式可以方便专业人士查找角色、载体、道具及环境模型。这些模型都不便宜，但质量高。

8.3 3D文件格式

这些年出现了许多 3D 文件格式，多到无法在这里全部介绍完。有些 3D 格式是某个 3D 软件自己用的，有些格式是用于在多个软件间交换数据的。有些格式是私有的，也就是说被某个公司或软件厂商完全控制，有些则是由一个工业小组开发的开放标准。有些格式是基于文本的，所以可读，有些则为了节省空间使用二进制格式。

3D 文件格式有三种分类：模型格式，用于表示单个模型；动画格式，用于动画的关键帧和角色定义；全功能格式，包含整个场景，以及其中的多个模型、变换层级结构、相机、光源及动画。我们会研究每种类型的格式，并重点研究适合 Web 应用的格式。

8.3.1 模型格式

单个模型的 3D 格式经常用来在不同的 3D 软件中交换数据。比如大部分 3D 软件都能导入和导出 OBJ 格式（接下来会介绍）。因为它语法简单且功能比较少，所以支持它很容易，因此很流行。然而，这也使得它所支持的功能不多。

1. Wavefront OBJ

OBJ 文件格式由 Wavefront 公司开发，它是业界最早也是支持最广泛的单一模型格式。它非常简单，只支持几何体（有顶点、法线及贴图坐标）。Wavefront 还引入了一个辅助的 MTL（Material Template Library，材质模板库）格式来支持给模型附上材质。

例 8-1 展示了一个基本的 OBJ 文件，它是从经典的“太空椅”模型中摘录出来的。这个模型我们在后面的章节会使用 Three.js 加载进来（效果见图 8-12）。OBJ 文件及相关代码在 models/ball_chair/ball_chair.obj 中，让我们看看它的语法。其中 # 字符用作注释定界符。这个文件包含一系列的声明。第一个声明是引用它所关联的材质库 MTL 文件。然后定义了几个几何体。这个摘录显示了用于定义名为 `shell` 的对象的一部分数据，它是太空椅的外壳。我们通过每行一条信息的方式，使用顶点位置、法线及贴图坐标数据来定义这个外壳。接下来是面的数据，同样是每个面一行。每个面的顶点通过类型为 `v/vt/vn` 的三个数来定义，其中 `v` 是顶点位置数据，`vt` 是纹理坐标数据，而 `vn` 是顶点法线数据。

例 8-1：一个 Wavefront OBJ 格式的模型

```
# 3ds Max Wavefront OBJ Exporter v0.97b - (c)2007 guruware

# File Created: 20.08.2013 13:29:52

mtllib ball_chair.mtl
#
# object shell
#
```

```
v  -15.693047 49.273174 -15.297686
v  -8.895294 50.974277 -18.244076
v  -0.243294 51.662109 -19.435429
... 其他顶点位置信息
vn -0.537169 0.350554 -0.767177
vn -0.462792 0.358374 -0.810797
vn -0.480322 0.274014 -0.833191
... 其他顶点法向量信息
vt 0.368635 0.102796 0.000000
vt 0.348531 0.101201 0.000000
vt 0.349342 0.122852 0.000000
... 其他纹理坐标信息
g shell
usemtl shell
s 1
f 313/1/1 600/2/2 58/3/3 597/4/4
f 598/5/5 313/1/1 597/4/4 109/6/6
f 313/1/1 598/5/5 1/7/7 599/8/8
f 600/2/2 313/1/1 599/8/8 106/9/9
f 314/10/10 603/11/11 58/3/3 600/2/2
... 其他面片定义
```

太空椅辅助的材质定义在 models/ball_chair/ball_chair.mtl 这个 MTL 文件中。它的语法非常简单，见例 8-2。一个材质通过 `newmtl` 语句来定义，它包含了使用 Phong 着色的一些参数，包括高光反射颜色和系数（`Ks`、`Ns` 和 `Ni` 关键字），漫反射颜色（`Kd`）、环境颜色（`Ka`）、发光颜色（`Ke`）和纹理贴图（`map_Ka` 及 `Map_Kd`）。MTL 中的纹理贴图经过几年的发展，包括了凹凸贴图、置换贴图、环境贴图及其他类型的纹理。在这个例子中，shell 纹理只定义了环境贴图和漫反射贴图。

例 8-2：OBJ 格式的材质定义

```
newmtl shell
    Ns 77.000000
    Ni 1.500000
    Tf 1.000000 1.000000 1.000000
    illum 2
    Ka 0.000000 0.000000 0.000000
    Kd 0.588000 0.588000 0.588000
    Ks 0.720000 0.720000 0.720000
    Ke 0.000000 0.000000 0.000000
    map_Ka maps\shell_color.jpg
    map_Kd maps\shell_color.jpg
...
```

2. STL

另一个基于文本的简单单一模型格式是 STL（StereoLithography），它被开发用于 3D 系统的快速原型、制造业及 3D 打印。STL 格式比 OBJ 更简单。这个格式只支持顶点几何，不支持法线、纹理坐标及材质。例 8-3 展示了一小段 STL 文件代码，它来自 Three.js 的一个例子（examples/models/stl/pr2_head_pan.stl）。可以打开 Three.js 中的 examples/webgl_loader_stl.html 来查看它运行时的样子。STL 是基于 WebGL 开发在线 3D 打印应用的不错格式，因为它能直接发到 3D 打印设备上。另外，它容易加载，且渲染速度快。

例 8-3：STL 文件格式

```
solid MYSOLID created by IVCON, original data in binary/pr2_head_pan.stl
  facet normal -0.761249 0.041314 -0.647143
    outer loop
      vertex -0.075633 -0.095256 -0.057711
      vertex -0.078756 -0.079398 -0.053025
      vertex -0.074338 -0.088143 -0.058780
    endloop
  endfacet
...
endsolid MYSOLID
```

STL 是如此简单和流行，GitHub 已经增加了直接查看它的支持（https://github.com/blog/1465-stl-file-viewing）。这个查看器是使用 Three.js 基于 WebGL 开发的。

STL 格式的技术细节可以参考维基百科（http://en.wikipedia.org/wiki/STL_%28file_format%29）。

8.3.2 动画格式

前面一节描述的格式只能用来表示静态模型数据。但大部分 3D 应用中的内容在屏幕上都是会动的（即被动画了）。有几种专门用来表示动态模型的格式，包括基于文本（因此对 Web 友好）的 MD2、MD5 及 BVH 格式。

1. id Software动画格式：MD2和MD5

你会时不时在 Web 上看到用于 id Software 的流行游戏 *DOOM* 和 *Quake* 的专有格式。MD2 及它后续的 MD5 格式，是用来定义角色动画的。这个格式被 id Software 所控制，他们很早就公布了这个格式的规范，并且有许多工具支持导入它们。

MD2 格式是为 *Quake II* 创建的，是一种二进制格式。它支持仅通过变形目标实现的基于顶点的角色动画。MD5（不要和在 Web 中广泛使用的加密散列函数“消息摘要”算法弄混了）是为 *Quake III* 开发的，它引入了蒙皮动画，并且是基于文本的可读格式。

MD2（http://tfc.duke.free.fr/coding/md2-specs-en.html）和 MD5（http://tfc.duke.free.fr/coding/md5-specs-en.html）规范的相关文档可以在网上找到。

要在 WebGL 应用中使用这些格式，我们可以自己编写一个加载器来直接读取它们，或者如果使用 Three.js 那样的库，我们可以使用转换工具。如果将 MD2 文件转成 JSON 格式，它看起来就像第 5 章中的例子那样，如图 5-11 所示。可以打开文件 examples/webgl_morphtargets_md2_control.htm 回顾一下，看一看它的源码，你会发现加载和解释 MD2 文件需要做许多事情。

Three.js 并没有在它的例子中包含 MD5 的加载器。但有一个不错的在线转换工具可以将 MD5 转成 Three.js JSON 格式，它是由 Klas（OutsideOfSociety）开发的，Klas 就职于瑞典的网络公司 North Kingdom（Find Your Way to OZ 的开发团队）。要查看一个转换后的模型，可以去 Klas 的博客。打开这个链接（http://oos.moxiecode.com/js_webgl/md5_

example/），你可以看到一个非常精细的怪物模型，可以控制它改变几种不同的姿势动画。

如果要转换你自己的 MD5 文件，可以打开这个链接（http://oos.moxiecode.com/js_webgl/md5_converter/），它允许你拖拽一个 MD5 文件到窗口中，然后输出 JSON 代码。

2. BVH：动作捕捉数据格式

动作捕捉（motion capture）是一种记录物体运动的方式，它在制作人体动画方面非常流行。在电影、动画、军事及运动应用中广泛使用。动作捕捉得到了开源格式的广泛支持，其中包括 Biovision Hierarchical Data（BVH）。BVH 由 Biovision 公司开发，用于表示人体的运动。BVH 非常流行，它基于文本，有很多工具支持导入和导出它。

开发者 Aki Miyazaki 做了一个将 BVH 数据导入到 WebGL 应用中的早期尝试。他的 BVH Motion Creator 是一个基于 Web 的 BVH 预览工具，它本身是基于 Three.js 开发的，如图 8-11 所示。借助这一工具，使用者可以上传 BVH 文件，预览它在简单角色上的动画效果。

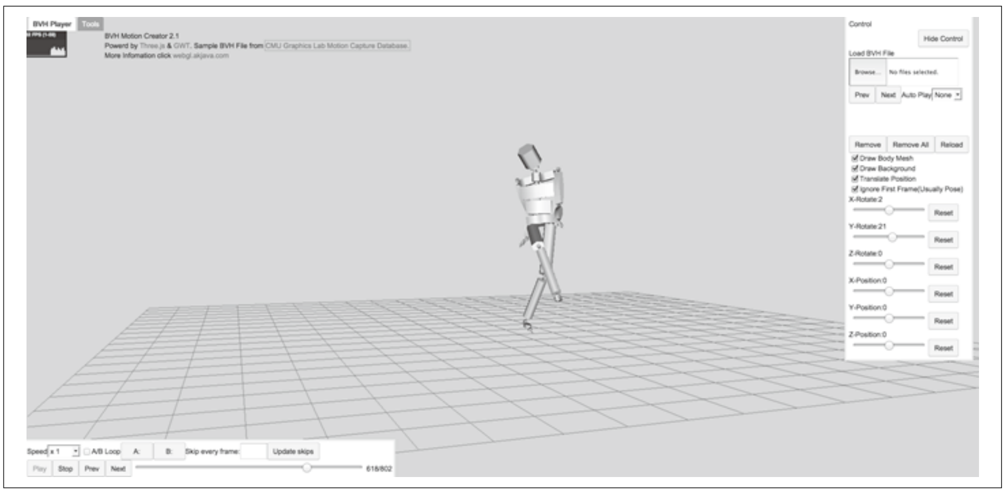


图 8-11：BVH Motion Creator（http://www.akjava.com/demo/bvhplayer/），一个 BVH 格式的预览工具

8.3.3 全功能的场景格式

多年来，业界开发了几种标准格式来支持表示整个 3D 场景，包括多个模型、变换层级结构、相机、光源和动画，以及所有在 3ds Max、Maya、Blender 这样全功能的软件中创造出来的东西。总之，这依旧是有待解决的复杂的技术问题，几种格式存活了下来并得到了广泛使用。但局势可能会有所改变，因为 WebGL 带动了新的需求，尤其是需要在应用间重用数据。本节中，我们会研究几种在 WebGL 中有可能使用的格式。

1. VRML和X3D

虚拟现实标记语言（Virtual Reality Markup Language，VRML）是最初用于 3D Web 的文本格式，它于 1994 年由发明家和理论家 Mark Pesce、硅谷图形公司（Silicon Graphics）Open Inventor 软件团队的成员，以及我所组成的小组创建。VRML 在 20 世纪 90 年代经历了几

次迭代，得到了业界以及一个非营利组织标准协会的支持。它的后续格式基于 XML，在 21 世纪初开发出来，名称改为了 X3D。尽管这些格式在 Web 应用中已经不再广泛使用，但大多数模型工具依然支持导入和导出它们。

VRML 和 X3D 定义了完整的场景、动画（关键帧、变形及蒙皮）、材质、光源，甚至还有脚本化了的、具有行为的交互式物体。例 8-4 展示了 X3D 语法，它创建了一个带有红色立方体的场景，当鼠标点击的时候，它会绕着 y 轴做 2 秒钟的 360° 旋转。X3D 的几何体、行为及动画都在一个 XML 文件中，直观且可读性好。直到今天，还没有一个开放的 3D 文件格式可以用如此简单和优美的语法实现同样的功能（我就自卖自夸了）。

例 8-4：X3D 的例子，当点击的时候红色立方体会旋转

```
 <?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "ISO//Web3D//DTD X3D 3.0//EN"
  "http://www.web3d.org/specifications/x3d-3.0.dtd">
<X3D profile='Interactive' version='3.0'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema-instance'
    xsd:noNamespaceSchemaLocation =
    ' http://www.web3d.org/specifications/x3d-3.0.xsd '>
<head>
... <!-- 此处为X3D文件的XML meta信息 -->
</head>
<!--
定义(DEF)节点索引:动画(Animation), 点击器(Clicker), 时间线(TimeSource), 变换矩阵(XForm)
-->
<Scene>
<!-- XForm ROUTE: [从Animation.value_changed到rotation ] -->
<Transform DEF='XForm'>
<Shape>
<Box/>
<Appearance>
<Material diffuseColor='1.0 0.0 0.0'/>
</Appearance>
</Shape>
<!-- Clicker ROUTE: [从touchTime到TimeSource.startTime ] -->
<TouchSensor DEF='Clicker' description='click to animate'/>
<!-- TimeSource ROUTEs:
[从Clicker.touchTime到startTime ] [从fraction_changed到Animation.set_fraction ] -->
<TimeSensor DEF='TimeSource' cycleInterval='2.0'/>
<!-- Animation ROUTEs:
[从TimeSource.fraction_changed到set_fraction ]
[从value_changed到XForm.rotation ] -->
<OrientationInterpolator DEF='Animation' key='0.0 0.33 0.66 1.0'
keyValue='0.0 1.0 0.0 0.0 0.0 1.0 0.0 2.1 0.0 1.0 0.0 4.2 0.0 1.0 0.0 0.0'/>
</Transform>
<ROUTE fromNode='Clicker' fromField='touchTime' toNode='TimeSource'
  toField='startTime'/>
<ROUTE fromNode='TimeSource' fromField='fraction_changed'
  toNode='Animation' toField='set_fraction'/>
<ROUTE fromNode='Animation' fromField='value_changed' toNode='XForm'
  toField='rotation'/>
</Scene>
</X3D>
```

VRML 的设计体现了交互式 3D 图形的许多关键概念，因此，你或许会认为它适合 WebGL 使用。但是，这个格式是在 JavaScript 和 DOM 都还没有出现前设计的，并且早于许多现今正在使用的硬件加速关键特性。基于这一点，依我个人浅见，VRML/X3D 过时了，不再适合实际使用。不过，它里面有许多可以借鉴到 WebGL 中的想法。

多年来，出现了许多 VRML 和 X3D 格式的内容。德国的 Fraunhofer 研究所还在继续 X3D 的研究，并创建了 X3DOM，一个使用 WebGL 显示 X3D 内容的查看器。更多关于 X3DOM 的信息，可以访问 http://www.x3dom.org/。

VRML（http://www.web3d.org/standardsvrml/）和 X3D（http://www.web3d.org/standardsx3d/）的规范可以在网上找到。

2. COLLADA：数字资源交换格式

2005 年左右，VRML 开始显得有些陈旧了，由包括 Sony Computer Entertainment、Alias Systems 和 Avid Technology 在内的一些公司组成了一个小组，一起开发一个 3D 数字资源格式，用于在游戏和交互式 3D 应用中交换数据。索尼的 Rémi Arnaud 和 Mark C. Barnes 领导开发了这个格式，命名为 COLLADA（COLLAborative Design Activity）。在第一版规范及相关公司支持完成后，这个标准的开发移交给了 Khronos 组织，该非营利性组织还开发了 WebGL、OpenGL，以及其他图形硬件及软件 API 的规范。

COLLADA 和 X3D 一样，是一个基于 XML 的全功能格式，可以表示整个场景以及多种不同类型的几何体、材质、动画及灯光。和 X3D 不同的是，COLLADA 的目标不是实现一个包含行为及运行时语义、面向最终用户的格式。事实上，它并没有明确的技术目标，而是试图完整保存从 3D 软件中可以输出的所有信息，使得它可以被后续的软件使用，或者被游戏引擎及开发环境导入，然后部署到最终的应用中。它的主要想法是，一旦 COLLADA 被业界广泛接受，各种 DCC 工具厂商就不需要编写插件导出为其他自定义格式，只要支持导出为 COLLADA，理论上任何其他软件都能导入。

例 8-5 展示了一个 COLLADA 场景的部分代码，我们将在本章后面使用 Three.js 加载它。这里简单讲解一下，COLLADA 文件结构的几个特点值得一提。首先，它是以库的形式来组织结构的，这些库有不同的类型，比如图片、着色器和材质。这些库首先在 XML 中进行定义，然后被其他需要的部分引用（比如材质定义中使用的图片）。其次，注意它会显式声明一些函数，而这些函数通常情况下都被看作内置函数，比如 Blinn 着色器。COLLADA 不对着色器和渲染模型做任何假设，它只是存储这些信息，让另一个工具拿到这些信息后再去做相应的处理。接着，我们发现网格顶点数据表示为一系列类型为 `float_array` 的元素。最终，通过引用了前面定义的几何体及材质（使用 XML 中的 `instance_geometry`、`bind_material` 及 `instance_material` 元素），这些网格组装成了一个用户可见的场景。

例 8-5：COLLADA 文件的结构，包括样本库、几何体及场景

```
<?xml version="1.0"?>
<COLLADA xmlns="http://www.collada.org/2005/11/COLLADASchema"
  version="1.4.1">
    <asset>
        <contributor>
            <authoring_tool>CINEMA4D 12.043 COLLADA Exporter
```

```
            </authoring_tool>
        </contributor>
        <created>2012-04-25T16:44:59Z</created>
        <modified>2012-04-25T16:44:59Z</modified>
        <unit meter="0.01" name="centimeter"/>
        <up_axis>Y_UP</up_axis>
    </asset>
    <library_images>
        <image id="ID5">
            <init_from>tex/Buss.jpg</init_from>
        </image>
          ... <!-- 其他图片的定义 -->
    </library_images>
    <library_effects>
        <effect id="ID2">
            <profile_COMMON>
                <technique sid="COMMON">
                    <blinn>
                        <diffuse>
                            <color>0.8 0.8 0.8 1</color>
                        </diffuse>
                        <specular>
                            <color>0.2 0.2 0.2 1</color>
                        </specular>
                        <shininess>
                            <float>0.5</float>
                        </shininess>
                    </blinn>
                </technique>
            </profile_COMMON>
        </effect>
        ... <!-- 其他滤镜的定义 -->
    <library_geometries>
        <geometry id="ID56">
            <mesh>
                <source id="ID57">
                    <float_array id="ID58" count="22812">36.2471
9.43441 -6.14603 36.2471 11.6191 -6.14603 36.2471 9.43441 -9.04828
36.2471 11.6191 -9.04828 33.356 9.43441 -9.04828 33.356 11.6191
-9.04828 33.356 9.43441
          ... <!-- 网格定义的其余部分 -->
...
    <!-- 以节点层级结构的形式定义场景 -->
    <library_visual_scenes>
        <visual_scene id="ID53">
                <node id="ID55" name="Buss">
                    <translate sid="translate">5.08833 -0.496439
-0.240191</translate>
                    <rotate sid="rotateY">0 1 0 0</rotate>
                    <rotate sid="rotateX">1 0 0 0</rotate>
                    <rotate sid="rotateZ">0 0 1 0</rotate>
                    <scale sid="scale">1 1 1</scale>
                    <instance_geometry url="#ID56">
                        <bind_material>
                            <technique_common>
```

```
                    <instance_material
                    symbol="Material1" target="#ID3">
                        <bind_vertex_input
                                semantic="UVSET0"
                                input_semantic="TEXCOORD"
                                input_set="0"/>
                    </instance_material>
                </technique_common>
            </bind_material>
        </instance_geometry>
    </node>
... <!-- 其余的场景定义 -->
```

经过了诞生初期高度的热情及广泛的厂商支持后，对 COLLADA 的支持开始衰退了。从大概 2010 年开始，流行 DCC 工具中导出这个格式的插件的开发几乎停止了。最近，COLLADA 又被重新重视起来，主要原因是对 WebGL 的支持的需求——WebGL 中缺乏内置的文件格式（后面会更详细地介绍）。随后出现了一个新的开源项目 OpenCOLLADA（https://www.khronos.org/collada/wiki/OpenCOLLADA），它包括了支持 3ds Max 及 Maya 2010 年以后版本的导出工具，能导出干净并兼容的 COLLADA 格式。

尽管增强对 COLLADA 格式的支持对于 3D 内容制作流程是一大好处，但它有个问题。正如我们在前面例子中看到的，COLADA 非常冗长。这个格式是设计用来保存更多数据的，而不是为了快速下载和解析。因此 Khronos 组织开始开发一个新的格式，既保留了 COLLADA 中的优秀特性——能完全表示丰富的 3D 动画场景，又同时考虑了 Web 传输问题，这个格式是 glTF。

3. glTF：一个用于WebGL、OpenGL ES及OpenGL应用的新格式

随着 WebGL 的逐渐流行，它带来了一个需要 Web 开发者解决的问题，那就是如何从 3D DCC 工具中导出完整的信息到 WebGL 应用中。类似 OBJ 那样的单一网格文本格式足以表示一个物体，但它不包含场景图结构、光源、相机及动画。COLLADA 的功能完善，但是我们在前面一节中看到，它很冗长，而且它是 XML 格式的，需要大量 CPU 计算来解析并转成可以被 WebGL 渲染的数据结构。我们需要一个紧凑、适合 Web 的格式，它在渲染前只需要进行最少的额外处理，类似 3D 领域的 JPEG 格式。

在 2012 年的夏天，Fabrice Robinet（摩托罗拉工程师、Khronos COLLADA 工作组主席）开始开发拥有 COLLADA 功能的 3D 格式，它比 COLLADA 更紧凑，而且更适合 WebGL。最开始这个项目被称为 COLLADA2JSON，它的想法是将笨重的 XML 语法转成轻量级的 JSON。从此以后，它开始走上了属于自己的发展道路。Khronos COLLADA 工作组中的其他成员也加入进来了，包括我、COLLADA 的创造者 Remi Arnaud，以及 Patrick Cozzi（防御软件供应商 AGI 的工程师）。我们的工作从简单地转换和优化 COLLADA 格式，变成了从头设计一种新的格式，用于基于 OpenGL 的 Web 及移动应用。于是 glTF（Graphics Library Transmission Format，图形库传输格式）诞生了。

glTF 以 COLLADA 的全功能特性为出发点，但它是一种全新的格式。COLLADA 的特性被用作支持何种图形特性的参考，但细节则完全不同。glTF 使用 JSON 来描述场景结构及高层信息（比如相机和光源），使用二进制格式来描述详细的数据，比如顶点、法线、颜色和动画。glTF 的二进制格式被设计为能直接加载到 WebGL 的缓冲中（比如 `Int32Array`

和 `FloatArray` 这样的类型数组）。加载一个 glTF 文件的流程可以简单地描述如下：

(1) 读取一个简单的 JSON 包装文件；
(2) 通过 Ajax 加载外部二进制文件；
(3) 创建一些类型数组；
(4) 调用 WebGL 绘制内容的方法来渲染。

当然，实际上这样看起来更复杂一点。但它比下载和解析 XML 文件、将 JavaScript `Number` 类型转成类型数组高效多了。glTF 可以保证文件体积小且加载速度快，这些都是创建高性能 Web 及移动应用中的关键因素。

例 8-6 展示了一个典型 glTF 场景的 JSON 语法，这个场景就是著名的 COLLADA 鸭模型。注意，它的语法结构类似 COLLADA，首先出现的是库，在最后定义场景图的结构，里面会引用这些库。但也就只有这些相同之处了。glTF 省掉了运行时非必需的信息，取而代之通过定义结构来让 WebGL 和 OpenGL ES 快速加载。glTF 定义了属性（顶点位置、法线、颜色、纹理坐标等）的详细信息，这些信息用于在可编程着色器中渲染物体。使用这些属性信息，glTF 应用可以如实渲染任意网格，即便它自己没有复杂的材质系统。

除了 JSON 文件，glTF 还会引用一个或多个存储了丰富数据（比如网格和动画的顶点数据）的二进制文件（后缀是 .bin），这些二进制文件中的数据结构被称为*缓冲*（buffer）和*缓冲视图*（buffer view）。使用这种方式，我们可以流式下载、增量下载，或者一次性加载 glTF 内容，这取决于应用需求。

例 8-6：glTF JSON 文件格式的例子

```
{
    "animations": {},
    "asset": {
        "generator": "collada2gltf 0.1.0"
    },
    "attributes": {
        "attribute_22": {
            "bufferView": "bufferView_28",
            "byteOffset": 0,
            "byteStride": 12,
            "count": 2399,
            "max": [
                96.1799,
                163.97,
                53.9252
            ],
            "min": [
                -69.2985,
                9.92937,
                -61.3282
            ],
            "type": "FLOAT_VEC3"
      },
... 此处省略其他顶点属性
        "bufferViews": {
            "bufferView_28": {
```

```
            "buffer": "duck.bin",
            "byteLength": 76768,
            "byteOffset": 0,
            "target": "ARRAY_BUFFER"
        },
        "bufferView_29": {
            "buffer": "duck.bin",
            "byteLength": 25272,
            "byteOffset": 76768,
            "target": "ELEMENT_ARRAY_BUFFER"
        }
    },
    "buffers": {
        "duck.bin": {
            "byteLength": 102040,
            "path": "duck.bin"
        }
    },
    "cameras": {
        "camera_0": {
            "aspect_ratio": 1.5,
            "projection": "perspective",
            "yfov": 37.8492,
            "zfar": 10000,
            "znear": 1
        }
    },
... 其他高级物体，如材质和灯光
... 最后是场景图(scene graph)
"nodes": {
    "LOD3sp": {
        "children": [],
        "matrix": [
           ... 矩阵数据
        ],
        "meshes": [
            "LOD3spShape-lib"
        ],
        "name": "LOD3sp"
    },
```

尽管 glTF 设计的重点是为 OpenGL 提供紧凑和高效的数据格式，但设计者还是做出了一定的折中，保留了 DCC 工具所创建的部分必备 3D 数据，比如动画、相机和光源。当前版本的 glTF（1.0）支持以下特性。

- 网格

  多边形网格由一个或多个几何基元组成。网格在 JSON 文件中定义，然后引用一个或多个包含顶点数据的二进制数据文件。

- 材质和着色器

  材质可以表示为高层的通用结构（Blinn、Phong、Lambert），或者在 GLSL 顶点着色器和片段着色器中实现，作为外部文件被 glTF 文件引用。

- *光源*

  通用光源类型（定向光、点光源、聚光灯和环境光），可以在 JSON 文件中用高层结构表示。

- *相机*

  glTF 定义了通用的相机类型，比如透视相机和正交相机。

- *场景图结构*

  场景使用一个层级图来表示，有许多节点（即网格、相机和光源）。

- *变换层级*

  场景图中的每个节点都有一个关联的变换矩阵。每个节点都可以包含子节点，子节点继承父节点的变换信息。

- *动画*

  glTF 定义了用来表示基于关键帧、蒙皮及变形动画的数据结构。

- *外部媒体*

  图片和视频可以用作纹理贴图的外部文件，使用 URL 来引用。

glTF 项目尽管是在 Khronos 组织的赞助下开发的，但它完全公开，允许任何人贡献代码。在 GitHub 中有一个源码仓库，其中包含了可以用的查看器、例子及规范本身。glTF 团队信奉的理念是功能需要先在代码中实现，然后才写到规范中。该团队已经开发了四个独立的 glTF 查看器，其中一个可以在 Three.js 中使用（我们待会将看到）。想要获取更多信息，请访问 Khronos glTF 的主页（http://gltf.gl/）。

4. Autodesk FBX

还有一个全功能场景格式值得一提，至少要顺带介绍一下。FBX 格式是一种文本格式，归属于 Autodesk 公司。最早它由 Kaydara 开发，用于 MotionBuilder 中。Autodesk 收购 Kaydara 后，它开始在它的多个产品中使用 FBX 格式。就这样，FBX 变成了 Autodesk 产品（3ds Max、Maya 和 MotionBuilder）内部交换数据的标准。

FBX 是一个富格式，支持许多 3D 和动画数据类型。和本章介绍的其他格式不同，FBX 是私有的，被 Autodesk 完全控制。Autodesk 编写这个格式的文档，并提供 C++ 和 Python 的 SDK 来读写 FBX 文件。但这些 SDK 需要产品许可证，有些可能会很贵。目前有一些不使用 SDK 开发的 FBX 导入导出工具，比如 Blender，但不清楚这样使用在 FBX 授权协议中是否合法。

因为这个格式是私有的，而且许可协议模糊，所以最好不使用 FBX。但另一方面，它是在业界顶尖工具中所使用的非常强大的技术，因此值得一看。想要获取更多信息，请访问 FBX 的主页（http://www.autodesk.com/products/fbx/overview）。

8.4　加载3D内容到WebGL应用中

还记得 WebGL 只是一个绘图库吧，它内部并没有多边形网格、材质、光源以及开发者在 3D 图形中用到的任何高层结构的概念。WebGL 只知道三角形和数学，因此 WebGL 没有

自己的文件格式并不奇怪。而且它也没有对本章之前介绍的所有格式的内建支持。为了将 3D 文件加载到你的 Web 应用中，你需要自己编写代码，或者使用相关的库。

值得高兴的是，Three.js 有许多加载常见格式的示例代码，包括 OBJ、STL、VRML 和 COLLADA 等。不过这些加载器代码仅仅只是示例代码，它们之间差别很大，有些加载器非常健壮，有些则不完善且有缺陷。Three.js 还定义了针对它自己的文件格式。它使用了基于 JSON 的清晰文本格式，还使用了二进制文件来压缩体积和提升加载速度，这和 glTF 很类似。它甚至还有一个基于 JSON 的、能包含多个物体的完整场景格式。不过这个格式还是实验性的，在我看来它还不能在正式产品中使用。

长话短说，你应该将 WebGL 中的 3D 内容制作流程当作一场探险。尽管我们最终会到达目的地，但在这条路上有许多曲折和意外。让我们开始这个探险吧，在本章的剩余部分，我们将会研究如何使用 Three.js 将内容加载到 WebGL 应用中。

8.4.1 Three.js JSON格式

Three.js 核心包中定义了自己的文件格式，它和 OBJ 功能类似，用于加载网格。但和 OBJ 不同的是，这个格式是基于 JSON 的，所以它解析后能很方便地为 Three.js 使用。

在本书编写的时候，还没有多少工具支持导出 Three.js JSON 格式。Three.js 团队写了一个 Blender 的导出器，所以这是一个可行的方式。事实上，如果你需要从各种不同格式导出到 Three.js JSON，这是一个不错的方式，因为 Blender 支持导入多种格式。如果你不喜欢用 Blender，另一种方式是使用 OJB 文件来转换。Three.js 有一个将 OBJ 文件转成 Three.js JSON 格式的工具，它是用 Python 写的，我们在下个例子中会用到它。

打开本书示例文件 Chapter 8/pipelinethreejsmodel.html。你可以看到一个经典的太空椅模型，就是一个中间有个大垫子的卵形椅子。使用鼠标左键来旋转模型，使用滚轮或触摸板来放大缩小，如图 8-12 所示。



图 8-12：一个 Wavefront OBJ 格式的文件，转成了 Three.js JSON 格式，并通过 `THREE.JSONLoader` 进行加载；这个经典的球形椅子模型来自 Turbosquid（http://www.turbosquid.com/FullPreview/Index.cfm/ID/761919），由 Luxxeon（http://luxxeon.deviantart.com/）创建

这个场景中的阴影和光线都是固定的，用来提供一个好看的背景，但模型都来自 OBJ。从 Turbosquid 下载这个美妙的模型后，我通过运行 OBJ 转换工具来创建能被 Three.js 加载的 JSON 文件。

转换工具在 Three.js 项目的 utils 子目录下。运行如下命令来进行模型转换：

```
python <path-to-three.js>/utils/exporters/convert_obj_three.py -i ball_chair.obj
-oball_chair.js
```

它会输出 ball_chair.js 文件。让我们看看文件的 JSON 语法，例 8-7 是它的部分代码。在一些描述版本号及其他信息的元数据后，就是内容部分。首先，有一些材质的定义。它们看起来应该很熟悉，因为是从 OBJ、MTL 文件转换过来的，我们在例 8-2 中已经介绍过。在那之后是网格定义，占据了文件的大部分内容。毫无疑问，这些 JSON 数组定义了顶点位置、法线、纹理坐标以及面片。一旦 Three.js 有了 JSON 中的这些信息，它就能轻松渲染我们前面看到的网格。

例 8-7：Three.js JSON 格式的例子

```
{

    "metadata" :
    {
        "formatVersion" : 3.1,
        "sourceFile"    : "ball_chair(blender).obj",
        "generatedBy"   : "OBJConverter",
        "vertices"      : 12740,
        "faces"         : 12480,
        "normals"       : 13082,
        "colors"        : 0,
        "uvs"           : 15521,
        "materials"     : 4
    },

    "scale" : 1.000000,

    "materials": [   {
    "DbgColor" : 15658734,
    "DbgIndex" : 0,
    "DbgName" : "shell",
    "colorAmbient" : [0.0, 0.0, 0.0],
    "colorDiffuse" : [0.588, 0.588, 0.588],
    "colorSpecular" : [0.72, 0.72, 0.72],
    "illumination" : 2,
    "mapAmbient" : "shell_color.jpg",
    "mapDiffuse" : "shell_color.jpg",
    "opticalDensity" : 1.5,
    "specularCoef" : 77.0
    },

... 其他材质定义

    "vertices": [-1.569305,4.927318,-1.529769,-0.889529,
```

```
... 其他顶点数据

    "morphTargets": [],

    "morphColors": [],

    "normals": [-0.53717,0.35055,-0.76718,-0.46279,0.35837,

... 其他法向量、颜色和纹理坐标数据

    "faces": [43,312,599,57,596,0,0,1,2,3,0,1,2,3,43,597

... 其他面片数据

}
```

现在让我们看看加载这个模型的代码。Three.js 并没有自带模型查看器，所以我们需要自己开发。但它非常容易（至少对于开发一个简单的模型查看器来说）。我们将把这个例子分成两个代码清单，一个创建场景和加载模型，另一个设置包括光源、背景及相机控制的场景环境。场景创建和加载模型的代码如例 8-8 所示。

例 8-8：加载一个 Three.js JSON 格式的模型的代码

```
function loadModel() {
    // 球形椅子模型由Luxxeon提供
    // http://www.turbosquid.com/FullPreview/Index.cfm/ID/761919
    // http://www.turbosquid.com/Search/Artists/luxxeon
    // http://luxxeon.deviantart.com/

    var url = "../models/ball_chair/ball_chair.json";

    // 卵形椅子模型由Luxxeon提供
    // http://www.turbosquid.com/FullPreview/Index.cfm/ID/738230
    // http://www.turbosquid.com/Search/Artists/luxxeon
    // http://luxxeon.deviantart.com/
    // var url = "../models/egg_chair/eggchair.json";

    var loader = new THREE.JSONLoader();
    loader.load( url, function( geometry, materials ) {
        handleModelLoaded(geometry, materials) } );

}

function handleModelLoaded(geometry, materials) {

    // 创建一个新的多面材质
    var material = new THREE.MeshFaceMaterial(materials);
    var mesh = new THREE.Mesh( geometry, material  );

    // 开启阴影
    mesh.castShadow = true;

    // 如果物体模型不在中心,将其移到原点
    geometry.computeBoundingBox();
    center = new THREE.Vector3().addVectors(geometry.boundingBox.max,
```

```
        geometry.boundingBox.min).multiplyScalar(0.5);
    mesh.position.set(-center.x, 0, -center.z);
    scene.add( mesh );

    // 基于几何形状的尺寸找到一个合适的相机位置
    var front = geometry.boundingBox.max.clone().sub(center);
    //camera.position.set(0, geometry.boundingBox.max.y / 2,
        geometry.boundingBox.max.z * 8);
    camera.position.set(0, front.y, front.z * 5);

    if (orbitControls)
        orbitControls.center.copy(center);
}

function createScene(container) {

    // 创建一个新的Three.js场景
    scene = new THREE.Scene();

    // 添加一个相机,以便我们观察整个场景
    camera = new THREE.PerspectiveCamera( 45, container.offsetWidth /
        container.offsetHeight, 1, 4000 );
    camera.position.z = 10;
    scene.add(camera);

    // 灯光
    createLights();

    // 地面
    if (addEnvironment)
        createEnvironment();

    // 模型
    loadModel();
}
```

首先，`createScene()` 函数创建了一个空的 Three.js 场景；然后，通过辅助函数（我们待会将会介绍），它创建一个相机，以及光源和背景。还记得吗，这些单一模型的格式是不包含相机和光源的，所以我们需要自己创建它们。

接下来，我们调用 `loadModel()` 函数来加载模型。它使用了 Three.js 内建的类 `THREE.JSONLoader`，这个类会将解析好后的 JSON 转成可以使用的 Three.js 几何体。我们调用加载器的 `load()` 方法，参数是模型的 URL 和一个回调函数。回调函数 `handleModelLoader()` 承担了少量的工作。在成功解析 JSON 后，Three.js 创建了一个几何对象并调用我们的回调函数。然后需要我们来创建材质，我们使用特殊的材质类型 `THREE.MeshFaceMaterial`。这个材质包含了几种材质：JSON 格式支持在几何体的每一面都使用不同的材质。我们使用回调函数第二个参数的不同材质来创建 `MeshFaceMaterial` 类型的对象。

现在我们的网格可以渲染了，我们将它添加到场景中。而且我们还加入了一些其他效果。我们想要有阴影，所以设置网格的 `castShadow` 属性为 `true`。我们想让网格在相机控制器中摆好位置，所以将它放到了原点处。我们可以通过调用 Three.js 的 `getBoundingBox()` 函数

来得到网格的中心。我们还通过边界框（bounding box）来计算合适的相机位置，将相机放在边界框的前上方。

例 8-9 给出了创建一个通用模型查看器的部分代码。首先，我们的渲染循环包含了一个对 headlight（一个白色的定向光源）的旋转，它永远从相机当前位置照射到场景中央。这样，我们就能以任意角度来查看模型。

我们希望有良好的阴影效果来提升视觉体验，所以在创建渲染器和场景光源的时候设置了必要的 Three.js 的 `shadow` 属性，请分别查看 `createRenderer()` 函数和 `createLights()` 函数。最后，我们需要一个地板来投射阴影，所以我们在 `createEnvironment()` 函数中创建它。

用于显示太空椅模型的代码可以作为样板代码：创建一个背景，创建一些默认的光源，创建一个相机，加载模型，在相机移动的同时保持光源的合适的朝向。这些步骤可以用来查看任意基本模型。

然而，这种代码的组织方式并不适合在应用中重用。我们将在下一章中解决这个问题，我们会开发一个通用的模型查看器类。但现在的关键问题在于：在 Three.js 中加载一个 OBJ 格式的单一模型文件原本就不复杂。

例 8-9：设置 JSON 模型查看器的背景和场景光照

```
function run() {
    requestAnimationFrame(function() { run(); });

        // 更新相机控制器
        orbitControls.update();

        // 调整headlight的位置,使其指向模型
        headlight.position.copy(camera.position);

        // 渲染场景
        renderer.render( scene, camera );

}

var shadows = true;
var addEnvironment = true;
var SHADOW_MAP_WIDTH = 2048, SHADOW_MAP_HEIGHT = 2048;

function createRenderer(container) {
    // 创建Three.js渲染器并将其添加到画布中
    renderer = new THREE.WebGLRenderer( { antialias: true } );

    // 开启阴影
    if (shadows) {
        renderer.shadowMapEnabled = true;
        renderer.shadowMapType = THREE.PCFSoftShadowMap;
    }

    // 设置视口尺寸
    renderer.setSize(container.offsetWidth, container.offsetHeight);
```

```
    container.appendChild(renderer.domElement);
}


function createLights() {

    // 灯光设置
    headlight = new THREE.DirectionalLight;
    headlight.position.set(0, 0, 1);
    scene.add(headlight);

    var ambient = new THREE.AmbientLight(0xffffff);
    scene.add(ambient);

    if (shadows) {
        var spot1 = new THREE.SpotLight(0xaaaaaa);
        spot1.position.set(0, 150, 200);
        scene.add(spot1);

        spot1.shadowCameraNear      = 1;
        spot1.shadowCameraFar      = 1024;
        spot1.castShadow            = true;
        spot1.shadowDarkness        = 0.3;
        spot1.shadowBias = 0.0001;
        spot1.shadowMapWidth = SHADOW_MAP_WIDTH;
        spot1.shadowMapHeight = SHADOW_MAP_HEIGHT;
    }
}

function createEnvironment() {
    // 地面
    var floorMaterial = new THREE.MeshPhongMaterial({
            color: 0xffffff,
            ambient: 0x555555,
            shading: THREE.SmoothShading,
        });
    var floor = new THREE.Mesh( new THREE.PlaneGeometry(1024, 1024), floorMaterial);

    if (shadows) {
        floor.receiveShadow = true;
    }

    floor.rotation.x = -Math.PI / 2;
    scene.add(floor);
}
```

8.4.2　Three.js的二进制格式

Three.js 定义了一个更紧凑的格式，它优化了网格的加载，是 JSON 格式的一个替代。这个二进制格式包含两个文件：一个是小巧的 JSON 外壳，用于定义网格的高层属性（例如材质列表）；一个是二进制文件（.bin 后缀），包含了顶点和面片的数据。

我们可以使用 Three.js 的 OBJ 转换工具来创建 Three.js 二进制文件，只需要简单地在命令行中使用 -t 参数：

```
python <path-to-three.js>/utils/exporters/convert_obj_three.py -i
ball_chair.obj -o ball_chair_bin.js -t binary
```

使用上面的命令来创建 ball_chair_bin.js 文件。来看看它的内容，JSON 和之前文本格式的版本很类似，只是所有网格的数据都被移到了二进制文件中，这些网格数据在 JSON 内通过 buffers 属性来引用：

```
"buffers": "ball_chair_bin.bin"
```

注意文件大小的区别。二进制文件格式（JSON 加上 .bin 文件）的大小大概是纯 JSON 版本的一半。要查看二进制文件动画的效果，请打开示例文件 Chapter 8/pipelinethreejsmodelbinary.html。模型和之前的效果是一样的，如图 8-12 所示。在 Three.js 中加载二进制格式，我们只需要修改一行，将类 THREE.JSONLoader 改成 THREE.BinaryLoader，参见例 8-10。

例 8-10：使用 Three.js 二进制格式来加载模型

```
function loadModel() {
    // 球形椅子模型由Luxxeon提供
    // http://www.turbosquid.com/FullPreview/Index.cfm/ID/761919
    // http://www.turbosquid.com/Search/Artists/luxxeon
    // http://luxxeon.deviantart.com/

    var url = "../models/ball_chair/ball_chair_bin.json";

    // 卵形椅子模型由Luxxeon提供
    // http://www.turbosquid.com/FullPreview/Index.cfm/ID/738230
    // http://www.turbosquid.com/Search/Artists/luxxeon
    // http://luxxeon.deviantart.com/
    // var url = "../models/egg_chair/eggchair.json";

    var loader = new THREE.BinaryLoader();
    loader.load( url, function( geometry, materials ) {
        handleModelLoaded(geometry, materials) } );

}
```

8.4.3 使用Three.js来加载COLLADA场景

Three.js 十分重视通过 OBJ 和它自身的 JSON 这样的单一模型格式来加载高质量的模型。迄今为止也都还不错，但它在很多使用场景下就不能胜任了。如果我们需要加载一个包含多个物体的场景，并保留它的变换层级结构以及相机、灯光、动画等其他有用的东西，就需要使用支持这些功能的格式。不然的话，我们就必须按顺序一个一个导入模型，整理，照亮，并手工进行场景动画。（不幸的是，在今天的 WebGL 开发中这样的景象经常出现，不过它在慢慢地改善。）

我们之前讨论过，COLLADA 是一种表现全场景数据的好格式。它支持我们需要的功能，并且有几个 3D 软件支持导出它。有了 COLLADA，就可以让设计师来创建复杂的场景，

包括建模、贴图、布置灯光及动画等，然后导出给 WebGL 使用，而不需要程序员手动来设置。COLLADA 的主要目的是将创意设计交给设计师。但是，它也有个缺点，就是使用了加载缓慢、笨重不堪的 XML 格式。不过，对于我们为了加载和查看完整的场景这一用途来说，它是一个好格式。

打开示例文件 Chapter 8/pipelinethreejsdaescene.html，你会看到一个漂亮的游戏背景艺术，有一片废墟和被遗弃的汽车，如图 8-13 所示。

这个例子加载了一个 COLLADA 场景，它由几个物体组了一个层级结构。我们通过一行加载调用代码来加载 COLLADA 文件。Three.js 的 COLLADA 加载器知道如何创建整个物体层级结构，包括各种相机、动画、光源等，而不需要我们参与。加载的回调做了一些额外的工作，它查找相机及光源，如果没有找到的话就设置为默认，仅此而已。显然我们不再需要为了将每个单独的物体摆放合适而手工固定它们的位置、方向以及缩放。比较一下 Three.js 项目中的典型示例场景代码，你会发现许多手工的数字。使用 COLLADA 是一种全新的体验。

让我们来看看加载 COLLADA 场景的代码，如例 8-11 所示。这个例子只展示了针对加载 COLLADA 场景的代码和对应的回调函数。

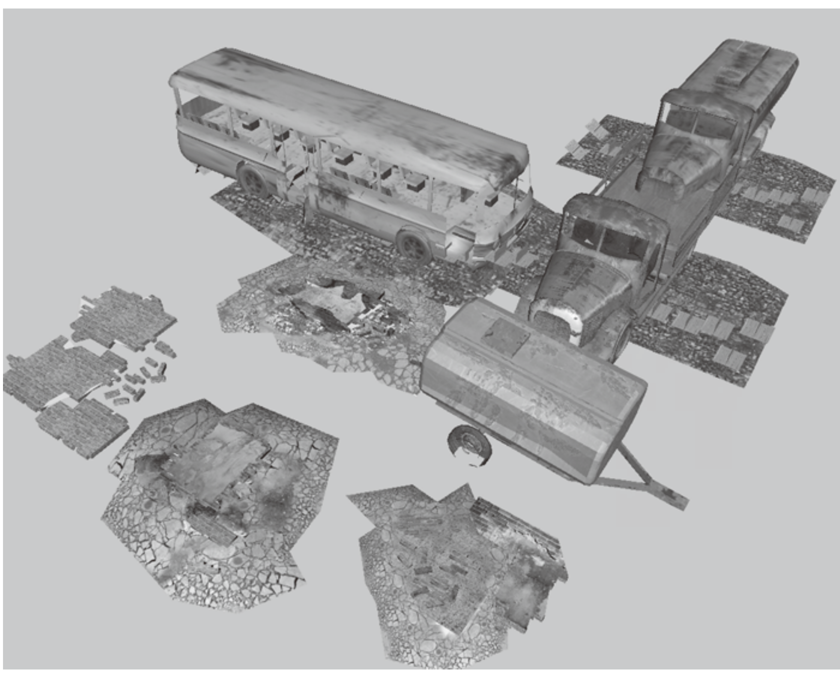


图 8-13：一个游戏场景地图素材，有层级结构和材质，使用 `THREE.ColladaLoader` 来加载 COLLADA 格式；该素材来自 Turbosquid（http://www.turbosquid.com/FullPreview/Index.cfm/ID/668298），由 ERHLN 创建（http://www.turbosquid.com/Search/Artists/ERLHN）

例 8-11：使用 Three.js 来加载 COLLADA 场景

```
function loadScene() {
    // 废墟模型由ERLHN提供
    // http://www.turbosquid.com/FullPreview/Index.cfm/ID/668298
    // http://www.turbosquid.com/Search/Artists/ERLHN
    var url = "../models/ruins/Ruins_dae.dae";

    var loader = new THREE.ColladaLoader();

    loader.load( url, function( data ) {
        handleSceneLoaded(data) } );

}

function handleSceneLoaded(data) {
    // 将物体添加到场景中
    scene.add(data.scene);

    // 遍历寻找相机和灯光
    var result = {};
    data.scene.traverse(function (n) { traverseScene(n, result); });

    if (result.cameras && result.cameras.length)
        camera = result.cameras[0];
    else {
        // 基于场景尺寸寻找一个最佳的相机位置
        createDefaultCamera();
        var boundingBox = computeBoundingBox(data.scene);
        var front = boundingBox.max;
        camera.position.set(front.x, front.y, front.z);
    }

    if (result.lights && result.lights.length) {
    }
    else
        createDefaultLights();

    // 创建控制器
    initControls();
}

function traverseScene(n, result)
{
    // 遍历寻找相机
    if (n instanceof THREE.Camera) {
        if (!result.cameras)
            result.cameras = [];

        result.cameras.push(n);
    }

    // 遍历寻找灯光
    if (n instanceof THREE.Light) {
        if (!result.lights)
```

```
            result.lights = [];

        result.lights.push(n);
    }

}
```

loadScene() 函数使用 THREE.ColladaLoader 类来加载废墟。回调函数 handleSceneLoaded() 调用的时候，会被传递一个参数 data。参数 data 包含了一个 JSON 对象，JSON 对象中的有些属性是 COLLADA 解析后的内容。我们关注的是 data.scene，它是一个 THREE.Object 对象，包含加载后的整个场景层级结构。我们将它添加到最顶层的场景上，让 Three.js 来渲染它。

我们现在基本上可以查看场景了，但我们还想要锦上添花，增加一些用户体验。首先我们遍历加载后的场景，查找相机和光源。如果找到了相机，我们会使用所找到的第一个作为我们的初始查看相机；如果没有找到，我们会创建一个默认相机。如果场景中有光源，我们就使用它；如果没有，我们就创建一个默认的光照设置。我们使用对象的 traverse() 方法来遍历场景，它会递归访问对象及其后代节点，然后调用我们提供的回调函数。我们的回调函数 traverseScene() 通过使用 instanceof 操作符来测试对象类型是否是 THREE.Camera 和 THREE.Light 来查找相机和光源，并将找到的结果分别插入到数组 result.cameras 和 result.lights 中。

当场景中不包含任何相机时，我们会创建一个自己的默认相机。我们希望根据场景的大小摆放相机的位置。为了计算场景的大小，我们使用了辅助函数 computeBoundingBox()。这个函数遍历场景来计算包含边界框。当它找到一个几何体时，就使用 Three.js 内建的 bounding-box 方法来查找几何体的边界框，然后将它合并到整个场景的边界框中。这个函数有点长，所以这里并没有展示它的代码。

8.4.4　使用Three.js来加载glTF场景

glTF 展现了 3D 文件格式的新方式。它专门设计用在 Web 和基于 OpenGL 的移动应用中，具有原生缓冲及其他适合渲染的数据结构。同时，glTF 包含了许多常用的 3D 结构，而这些结构在 OpenGL ES 中并没有直接表现，比如材质、相机和灯光。glTF 的目标是创建一个紧凑的文件格式，能方便地在 Web 和移动应用中加载，同时还能表示实际开发中用到的 3D 数据类型。

目前已经有几个正在开发中的项目，用来在图形库和应用添加 glTF 支持。这其中也包括我为 Three.js 编写的 glTF 加载器。打开示例文件 Chapter 8/pipelinethreejsgltfscene.html。你可以看到类似于图 8-14 所示的效果。几个太空船在未来的城市上空巡航。这个场景的渲染效果很不错，有环境贴图和 Blinn 着色。里面有几个动画效果，其中包括了移动摄像头。使用下拉框来切换相机和加载不同的场景，点击 Animation 复选框来开始和停止动画。这个场景原本是在 3ds Max 中创建的。Fabrice Robinet 从 3DRT.com 上下载了这个 3ds Max 文件，导出为 COLLADA，然后运行转换工具转成了 glTF 文件。



图 8-14：使用还在开发中的实验性的 `THREE.glTFLoader` 类加载 glTF 场景（包括动画、场景图层级结构、材质、光源和相机）；这个加载器的源代码在 glTF GitHub 项目页面上（https://github.com/KhronosGroup/glTF），这个虚拟的城市场景来自 3DRT（http://3drt.com/store/free-downloads/33-sci-fi-skyscrapers-collection.html）

我照着 Three.js 示例中加载其他文件格式的加载器来设计 glTF 加载器。类 `THREE.glTFLoader` 继承自加载器的基类 `THREE.Loader`。它的 `load()` 方法用来解析 glTFJSON 文件；载入外部资源，如二进制缓冲、纹理和着色器；以及通过回调函数来返回结果。回调函数能够访问加载器创建的 Three.js 对象层级结构，因此可以轻松将它加载到场景中并开始渲染。

使用 glTF 格式的先期优势是十分可观的，至少和同等的 COLLADA 差不多。文件体积是 COLLADA 文本格式的一半，加载某些模型的速度提升了 80%。这有一部分归功于我们使用了 Three.js 中新的 `BufferGeometry` 类型，它允许我们用加载好后的类型数组数据（比如 `Int32Array` 和 `FloatArray`）来直接创建几何体，而不是使用普通的 JavaScript `Number` 类型的数组（不管怎样，它最后还是需要转成类型数组，才能使用 WebGL 渲染）。

8.5 小结

本章探索了如何为 WebGL 创建 3D 内容。在简要介绍创建流程后，我们介绍了 3D 内容创建工具，从业余到专业、从本地软件到浏览器中的集成环境都有介绍。

我们一睹了当今应用中使用的 3D 文件格式，尤其是那些适合 WebGL 在线使用的格式。这既包含了旧的标准，也包括了新的格式——glTF，它是专门为今天的 Web 和移动应用设计的格式。最后，我们学习了使用 Three.js 库来加载各种格式的详细例子，包括单个模型格式和整个场景。

尽管并没有一个在 Web 应用添加 3D 内容的首选方式，而且 WebGL 内容制作流程还很年轻并处在发展之中，但至少目前有几种可行的方式能够完成这项任务。

第 9 章

3D引擎和框架

Three.js 是一个非常棒的库，它让用 WebGL 渲染复杂 3D 内容这一艰巨任务变得简单，人人都能实现。如果没有 Three.js 这样的库，WebGL 开发者需要花费几个月的时间才能实现类似的效果。但尽管很强大，Three.js 依然有它的局限性。它能做好绘制，但仅此而已。对于其他所有的事情，你都需要自己完成。

比如你想开发一个购物应用，让用户能在购买前自定义汽车。Web 页面中显示车的 3D 模型，用户可以点击汽车的不同部分来改变颜色和样式。简单点击一下按钮，就会有动画效果平滑地从车的外部过渡到车的内部。如果仅使用 Three.js，你可能需要写成百上千行代码来实现这个应用。尽管有原始的工具箱，但它并没转成一系列更高层次的可重用组件。Three.js 是被设计用来作为场景图和渲染库的，但 3D 应用开发不仅仅是绘制图片。

自定义汽车的场景包含了通用 3D 开发事项：加载模型、通过名称或 ID 获取模型的各个部分、当点击某个部分的时候触发行为、修改相机视图，等等。这些设计模式在游戏、虚拟世界、建筑浏览、教育软件、模拟训练等大多数 3D 应用中很常见。如果你在开发专业级的 3D 应用，并且不想花时间在发明新方法以解决旧问题上，那你应该考虑使用高级的引擎或框架。

本章介绍 3D 应用框架的概念，并着眼于基于 WebGL 的方案。这些框架有许多都是基于 Three.js 开发的，所以如果你已经花费很多精力学习了 Three.js 图形学，就不需要再去学习其他全新的东西。本章后面会介绍我自己设计的一个框架——Vizi，我们将在后面几章中使用它来构建例子。这些框架中包含的概念都很通用，它们大多都适用于其中任意一个框架，此外它们在你决定开发自己的框架时也很有帮助。

9.1 3D框架概念

框架为开发者提供了内建的功能，并实现可靠、可重用的通用设计模式。它可以帮助我们节省时间，写出更好的应用（至少从理论上来说）。一个好的框架可以让我们不必重复造轮子，因为它借用了老练开发者的经验，让我们专注于自己手头上的工作。

9.1.1 什么是框架

框架并没有一个严格的定义。实际上，经常很难区分框架和库。它们都是设计用来节省时间的，提供了可重用的代码。此外，它们都掩盖了底层操作系统或平台的实现细节，为低级的服务提供了高级的接口。然而，有一些区别能显示出我们使用的是框架而不是库[1]。

- *抽象的层次*

  框架会比库在更高的抽象层次上工作。例如，3D 库或许支持角色动画的蒙皮，而 3D 框架则会将蒙皮后的网格及一系列动画手势封装起来，成为一个化身（avatar）。框架会根据用户的输入自动地在场景中移动化身，然后通过回调函数来通知我们。

- *默认行为*

  框架提供了默认行为，例如，当一个场景创建后，一个默认的相机就会放在里面一个已知的位置和角度。好的框架还会竭尽全力地让开发者覆盖默认的行为，以提供灵活性。

- *扩展性*

  框架加强了扩展性，允许第三方扩展的开发和自定义。最好的框架会一方面提供强大的内建组件，另一方面还允许扩展或彻底替换系统中一部分。

- *反向控制流*

  或许框架最显著的特征就是开发者并不能控制流程。开发者只是简单地提供回调函数，或者覆盖方法来实现应用程序的特殊功能。回想一下 WebGL 应用的典型页面创建过程：创建好场景、初始化渲染器、调用运行循环。使用一个 WebGL 框架后，开发者只需要提供场景创建的代码，由框架来完成剩余部分的搭建。

框架和库还有一个非技术性方面的区别：框架往往更极端。它被认为是双刃剑，像浮士德式的交易那样可以让我们快速开发并投向市场，但最终在项目结束前会偷走我们的灵魂。框架可以快速提供我们所需的 90% 功能——在开发早期让我们过度自信——然后在实现最后 10% 的时候变得异常复杂。框架很难调试和优化，因为我们使用了其他人的代码。如果你曾经使用过 Zend 或 Rails 那样的 Web 开发框架，会体会过类似的哀伤。因此，许多开发者干脆避免使用框架。相比之下，像 jQuery 那样非侵入式的库获得了开发者的支持，因为它提供了强大的功能，但没有带来麻烦。

> 开发者是有激情的生物，就像其他人一样。没有什么可以像关于良好的老式框架的争吵那样能激发开发者的热情了。如果你看到有人正在争吵，最好避而远之。对于在自己的项目中使用框架，我有种很强烈的感受，可以概括为：
>
> *我热爱框架……只要它是我的。*

注 1：基于维基百科条目中对软件框架的详细讨论。

不管你对框架的感受如何，你在构建任意大小的 3D 应用时都会面临本章所讨论的问题。你同样会面临一个选择：开发你自己的框架，或者使用现有的框架，抑或准备好写大量额外的代码。

9.1.2 WebGL框架需求

我们可以将浏览器看成一个 2D 应用框架。DOM 和 CSS 提供了一系列预定义的可视对象，由浏览器来进行渲染。应用开发由提供一些回调函数来构成，这些回调函数基于用户行为产生，比如点击一个按钮、页面已经加载完成等。当应用想要改变页面的外观或内容时，它设置一个或多个属性，由浏览器自动更新呈现。

但不幸的是，除了 CSS 3D 变换以外，浏览器预定义的对象都不能扩展到三维。从 HTML5 开始，浏览器架构的重点从预先定义可视对象（比如文本、滚动条、按钮等）转向了允许控制渲染和其他系统级别的功能。WebGL 和 Canvas 允许我们绘制我们想要的任何东西，但所有其他工作都需要我们自己动手。一旦进入 Canvas 元素的世界，我们就需要创建自己的场景图、事件模型、交互、行为、动画以及过渡——或者更好的方式是使用现成的框架来帮我们完成这些工作。

WebGL 应用对框架设计提出了独特的要求。除了面临着一大堆经典的 3D 特有问题外，还需要满足能够在基于浏览器的平台上工作这一需求。一个 WebGL 框架应该包含以下大部分功能。

- 环境设置

  框架检查是否支持 WebGL，然后创建绘图上下文以及其支持绘制的其他对象。它同时还增加了对 DOM 事件的监听，比如窗口缩放、鼠标和键盘输入、WebGL 上下文丢失及其他页面事件，然后在需要的时候分配到应用中。

- 能力检测和降级

  框架检测各种浏览器能力，然后提供可能的填充（polyfill）或提供降级，比如如果不支持 WebGL 就使用 2D Canvas 进行绘制。

- 默认场景创建

  框架创建一个空的场景，该场景可能包含一个默认的相机和默认光照。

- 模拟 / 运行循环

  框架提供运行循环，应用通过提供事件的回调函数和覆盖方法来实现应用特有的功能。框架或许还会确定一个严格的时钟概念或时间模型，应用必须遵守它来获得一致的行为。

- 图形和渲染

  框架提供对象来渲染图形。例如在基于 Three.js 的框架中，它意味着由框架来管理对 Three.js 对象的访问。

- 对象和事件模型

  框架指定了对象属性的统一模型，对象在层次结构或图中和其他对象的关系，以及对象如何通过事件、回调、访问方法来进行交互。

- 交互

  框架自动将鼠标及其他输入映射到场景中的具体对象，并在对象被点击、拖拽时通知应用程序。

- 导航/查看方法

  框架可能会提供一个或多个导航模型，它是指相机在场景中移动的高级模型（比如第一人称射击），用来处理碰撞和地形跟随，或者旋转相机来查看一个特定物体。还可能会有内建逻辑来切换相机和场景间的过渡。第一人称射击导航模型或许还经常定义在多人环境下使用一个化身来代表用户。

- 行为和动画

  框架会有预定义的行为，从简单的物体随时间旋转和变换，到由交互触发的复杂动画序列。

- 物理

  有些框架提供了丰富的物理模型：以一定速率在一个方向上移动，设置重力，检测物体间的碰撞等。

- 资源加载

  框架为程序员自动加载模型、纹理、视频和声音，并在资源载入完成后通知应用。强大的框架甚至包括客户－服务器端加载的方案，能够流式加载 3D 数据和动画，渐进式提供高精度的网格细节。

- 场景工具

  框架有对场景图控制的广泛支持。查询 API 会查找特定类型的对象，或者使用正则表达式或选择符来匹配 id，然后应用各种操作：修改材质属性、应用 3D 变换，增加或删除子节点，显示或隐藏物体。

- 内存管理

  尽管基于 JavaScript 的应用有自动垃圾清理机制，但复杂的应用需要注意内存是如何以及何时分配的。不然的话，垃圾回收清理会在不合时宜的时候发生，影响帧率，进而影响用户体验。有些框架提供了智能的内存管理服务，来帮助避免这些问题（在第 12 章有关于这个话题的更多讨论）。

- 性能支持/优雅降级

  框架或许会根据帧率和资源占用情况来自动调节分辨率或渲染质量，目标是提供一致的用户体验。

- 扩展机制

  好的框架不会限定开发者只使用内建的组件，它们允许进行扩展。对于一个 WebGL 框架来说，这意味着提供行为的回调、覆盖交互，最重要的是，可以自定义渲染来改变视觉外观。

这个列表很长，创建一个高质量的 3D 应用需要做很多事情，而框架会大有帮助。现在已经有几个使用 WebGL 的好框架，我们在下一节中进行介绍。

9.2　WebGL框架概况

WebGL 框架分为两类：游戏引擎和展示框架。游戏引擎一般更强大，但更难使用和掌握；而展示框架更适合创建相对简单的应用，比如嵌在页面中的、只有基本交互的模型。本节介绍了许多在本书编写时还在开发中的 WebGL 框架。

9.2.1　游戏引擎

如果你的目标是开发一个顶尖的 WebGL 游戏，你应该考虑使用这几年出现的诸多游戏引擎之一。游戏引擎和框架间的不同比较微妙。一般来说，游戏引擎比普通框架提供更多的功能。但另一方面，它们是为更加有经验的开发人员设计的。游戏开发包含复杂的技术，而游戏引擎反映了这一点。

有几个可选的 WebGL 引擎。它们之间的能力差别很大，对经验技能的要求也有很大不同。有些引擎是开源的，有些则不是。有些可以免费使用，而有些需要购买授权、缴纳托管费，或者是鼓励你通过它的销售网络来发布游戏。这里列出的游戏引擎没有一个使用 Three.js 来进行渲染，而是选择了控制整个流程。有一些权衡，需要在你评估项目引擎的时候考虑。

- playcanvas（http://www.playcanvas.com/）

  位于伦敦的 playcanvas 开发了一个功能丰富的引擎和基于云的编辑工具。该编辑工具的特色有：可以实时协同编辑场景，来支持团队开发；集成了 GitHub 和 Bitbucket；在社会媒体网站上一键发布。图 9-1 显示了运行中的 playcanvas 游戏。



图 9-1：基于 playcanvas 创建的第一人称射击游戏（http://www.playcanvas.com/）

- Turbulenz（http://biz.turbulenz.com/developers/）

  一个极其强大、开源且免费的游戏引擎，提供了可下载的 SDK。如果你想要在这个公司的网络（http://biz.turbulenz.com/developers/）上发布游戏，它会收取一些提成。Turbulenz 在 API 上很复杂，它提供了一大堆类，学习曲线陡峭。它明显是给经验丰富的游戏开发者使用的。

- Goo Engine（http://www.gootechnologies.com/）

  在本书编写的时候，这个引擎还处于内部测试阶段。它在网站上自夸有许多传统游戏引擎的功能，并且通过 WebGL 来实现跨平台。它的网站的技术和授权信息很少，但示例很漂亮，如图 9-2 所示。



图 9-2：一款基于 Goo Engine（http://www.gootechnologies.com/）开发的水中冒险游戏；图片来自 Pearl Boy

- Babylon.js（http://www.babylonjs.com/）

  微软最近加入了 WebGL 的浪潮，给了它一个巨大的推动。Babylon.js 是一个易于使用的引擎，在功能和易用性方面介于 Three.js 和专家型游戏引擎之间。示例网址展示了 Babylon.js 广泛的应用，从太空射击到建筑浏览都有。

- KickJS（http://www.kickjs.org/）

  由 Morten Nobel-Jørgensen 创建的开源游戏引擎和渲染库。这个项目由 Nobel-Jørgensen 的学术工作发展而来。它看起来开发不足且缺少支持，所以要用的话得小心。我在这里包括它，是因为在提过的所有引擎中，KickJS 是最遵循现代游戏引擎设计规范的（关于这个话题我们将在介绍 Vizi 的时候详细讨论）。如果没有其他选择了，它可以作为设计你自己的框架的重要参考。

可以看到，有许多潜在的 WebGL 游戏引擎可选。你或许甚至会使用游戏引擎来开发非游戏的应用。但要记住，游戏引擎的学习曲线陡峭，所以需要保证解决方案与问题相适应。对于简单点儿的视觉应用，你可以使用更简单的 3D 框架。下一节我们将对其中一些进行介绍。

9.2.2 展示框架

游戏只是可以基于 WebGL 开发的 3D 应用中的一种。对于很多非游戏应用，比如页面图形、电子商务产品展示或者科学可视化，使用游戏引擎类似于杀鸡用牛刀。展示应用通常只需要加载简单的场景到页面中，播放几个动画，并通过修改几个属性来响应用户输入。之前我们介绍过，即便是这些基本操作在 Three.js 中也需要大量额外的开发工作，所以我们使用框架来辅助开发。以下是几个可以考虑的通用 3D 展示框架。

1. tQuery

tQuery（http://jeromeetienne.github.io/tquery/）由 Jerome Etienne 创建。Jerome 管理着人气博客 Learning Three.js（http://learningthreejs.com/），其中包含了大量 Three.js 开发诀窍和技巧。

tQuery 模仿了 jQuery，它的想法是既有 Three.js 的能力又有 jQuery API 的易用性，也就是说能够使用非常简单的 API 来创建 Three.js 场景图。它使用了链式函数的编程风格，支持通过回调来实现高级的交互行为。使用 tQuery 可以省去大量 Three.js 的硬编码。或许将 tQuery 称为框架并不正确，因为它所模仿的 jQuery 的精神是无侵入的库。如果你是一个 Three.js 开发者，想少敲些键盘，你应该认真研究一下它。

例 9-1 展示了一个简单的代码段，它的功能是使用 tQuery 将一个圆环体放到页面中，和前几章使用 Three.js 的例子进行对比，你会看到框架如何让 3D 开发变得简单。

例 9-1：使用 tQuery 创建一个简单的场景

```
<!doctype html><title>Minimal tQuery Page</title>
<script src="tquery-bundle.js"></script>
<body><script>
    var world = tQuery.createWorld().boilerplate().start();
    var object = tQuery.createTorus().addTo(world);
</script></body>
```

Etienne 的设计理念可以大概总结为“让 3D 开发尽可能像 2D 开发那样”。Web 开发者已经了解 jQuery 了，给他们一个像 jQuery 那样的 API 去开发 3D 应用，他们立刻就会具有生产力。这个逻辑很难反驳。

2. Voodoo.js

居住在西雅图的 Brent Gunning 致力于让人人都能创建 3D 应用。在因 WebGL 的能力而兴奋却失望于其编程难度的前提下，于是 Brent Gunning 创建了 Voodoo.js（http://www.voodoojs.com/）。Voodoo.js 的目标是让创建 3D 应用变得简单且容易集成到页面中。Gunning 在第一次发布 Voodoo.js 的时候在他的博客上做了如下总结：

> 在今天的 Web 中，3D 只是一个玩具，一个噱头。创建任何 3D 形式的内容都需要大量额外工作且几乎无法轻易重用。更糟糕的是，只因它多了一个维度，我们就将 3D 场景放到 Canvas 中，与 2D 内容隔离开来。这是设计上的噩梦，我想做点什么改进它。因此，我很高兴地宣布 Voodoo 的第一次公开发布，版本是 0.8.0 beta。

Gunning 的愿景不仅包括简单的拖拽开发，还包括一个可重用的生态系统，包括了对象、组件、视觉样式和主题。Voodoo.js 框架由一系列小巧的预制功能的类构成，包括模型的加

载和查看、基于鼠标的交互、几个可选配置。这个框架基于 Three.js 开发，因此理论上它应该很容易扩展和自定义新的对象类型。例 9-2 来自 Voodoo.js 首页，它只使用一个函数调用就创建了一个 3D 对象，然后插入到页面元素 `example2`。这不能再简单了。它的效果如图 9-3 所示。

例 9-2：使用 Voodoo.js 将一个 3D 对象插入到页面中

```
new VoodooJsonModel({
  elementId: 'example2',
  jsonFile: '3d/tree.json',
  offsetWidthMultiplier: 2.0 / 3.0,
  scale: 50,
  rotationX: Math.PI / 2.0,
  rotationY: Math.PI / 2.0
});
```

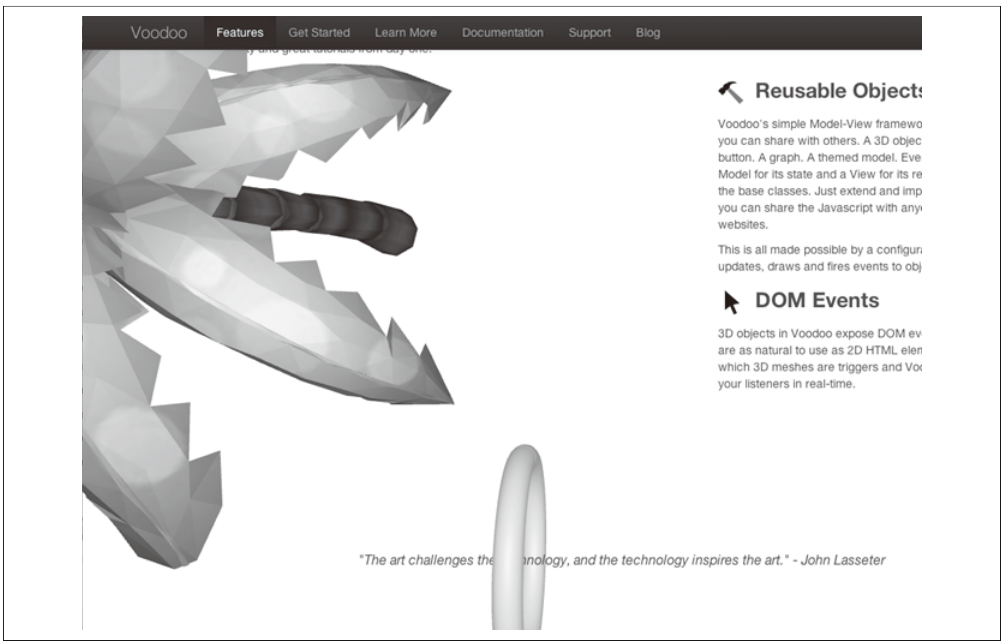


图 9-3：Voodoo.js 首页（http://www.voodoojs.com/），特点在于有几个内嵌的 3D 对象

3. PhiloGL

PhiloGL（http://www.senchalabs.org/philogl/）是数据可视化科学家 Nicolas Garcia Belmonte 在 Sencha 公司的实验室内创建的实验项目。PhiloGL 的目标是让 WebGL 开发尽可能简单和有趣，就像使用任何主流框架进行开发一样。Garcia 在介绍性博文（https://www.sencha.com/blog/）中描述了这一设计理念。尽管这个框架是实验性质的，但它值得一看。Sencha 公司开发了世界级的用户界面框架，十分精通如何使用 HTML5 创建高效的用户界面。例 9-3 展示了使用 PhiloGL 创建一个简单场景的代码。通过定义几个 JavaScript 对象，我们创建了包含一个带纹理球形的场景。PhiloGL 的网站上有几个可以工作的例子，包括来自 Learning WebGL（http://learningwebgl.com/blog/）的所有教程。

例 9-3：使用 PhiloGL 创建一个简单的 3D 场景

```
// 创建一个应用
PhiloGL('canvasId', {
    camera: {
        position: {
            x: 0, y: 0, z: -7
        }
    },
    scene: {
        lights: {
            enable: true,
            ambient: { r: 0.5, g: 0.5, b: 0.5 },
            directional: {
                color: { r: 0.7, g: 0.7, b: 0.9 },
                direction: { x: 1, y: 1, z: 1 }
            }
        }
    },
    textures: {
        src: ['moon.gif']
    },
    events: {
        onClick: function(e) {
            /* 在此编写事件响应 */
        }
    },
    onError: function() {
        alert("There was an error creating the app.");
    },
    onLoad: function(app) {
        // 初始化应用
        // 往场景中添加物体
        scene.add(moon)
        // 动画
        setInterval(draw, 1000/60);
        // 绘制场景
        function draw() {
            // 渲染月球
            scene.render();
        }
    }
});
```

9.3 Vizi：一个基于组件的用于可视化Web应用的框架

现在到了仔细研究基于框架的 3D 开发的时候。我们将使用一个足够通用的框架，以覆盖大量可能的使用场景。尽管万能的 3D 系统并不存在，但在应用中还是有许多通用的模式。据此我创建了 Vizi —— 我自己设计的一个 WebGL 框架。我将用它来开发接下来几章中的例子。本节通过深入探索基于框架开发的概念来介绍 Vizi。

9.3.1 背景和设计理念

正如 tQuery、Voodoo.j 及 PhiloGL 的开发者那样，我对 WebGL 开发的状况感到失望。我是 Mr.doob 的忠实粉丝之一，但我认为光有 Three.js 还不足以开发产品级质量的应用。我们这里讨论的大部分问题，其实在几年前的 3D 框架和游戏引擎中就已经解决了。底层的平台当然在这些年中已经进化了，但问题大体上还是一样的：载入场景内容、设置相机、绘制一些对象、根据计时器和用户输入来移动物体、重复上述步骤。

这几年游戏引擎的设计发生了变化。最近二十年游戏产业变得很有活力，可以说它触发了计算机历史中一些最大的创新，包括软件引擎的设计。最为显著的是它从基于类和继承的架构转向了基于组件和集成的架构（这或许看起来是技术上的区别，但它有极大的影响，我们稍后会看到）。基于前面介绍的许多 3D 开发项目，并换个新的视角基于当前游戏引擎的最佳实践，我决定做一个新的探索，Vizi 就是这样诞生的。

Vizi 的目标是让快速开发有趣的 3D 应用变得简单。在功能方面，Vizi 介于游戏引擎（如 playcanvas）和展示框架（如 Voodoo.js）之间。本章的产品配置场景很适合使用 Vizi，它是一个有多个交互对象的场景，基于用户输入动态更新，模型是按需加载的，有复杂的视角和基于相机的导航。我相信这些功能体现了 WebGL 开发最具优势的地方，因此这也是我最想要强化的设计之处。

图 9-4 展示了一个 Vizi 应用原型，是一个电子商务网站概念，虚拟车的展示。高精度的图片窗格在场景中间缓慢地旋转，像旋转木马一样。窗格在精巧的网格背景上投影。页面加载几秒钟后，一个完整的 3D 车模型被举上展示台。高清的车身上反射了背后的网格环境。点击窗格会让它向前居中放大，播放视频广告。2D 的用户界面元素在周围提供了访问其他信息及网站其他部分的渠道。这只是一个概念，但它描绘了 Vizi 的核心想法：混合使用 2D 和 3D 内容为电子商务及其他 Web 应用提供新型交互。

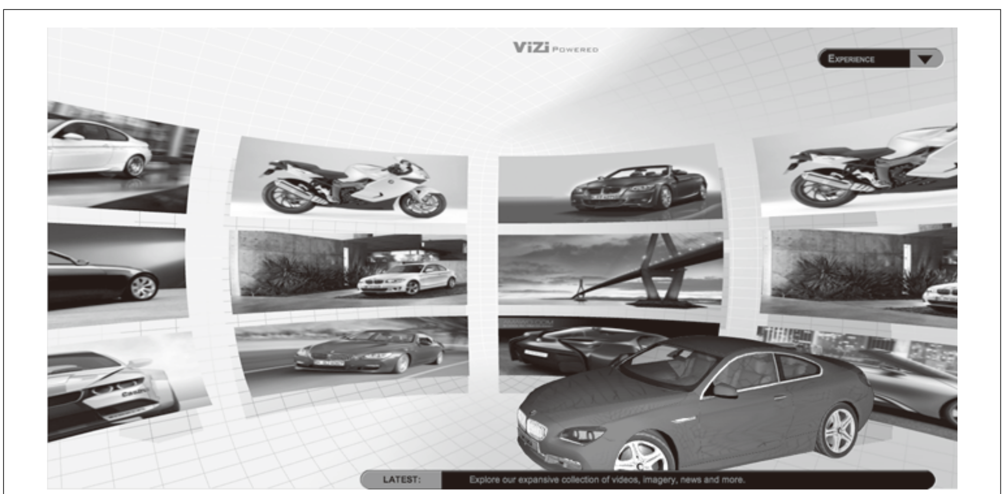


图 9-4：基于 Vizi 开发的概念车展示应用；车模型来自 be fast（http://www.turbosquid.com/Search/Artists/be-fast），视觉及环境设计来自 TC Chang（http://www.tcchang.com/）

9.3.2 Vizi架构

Vizi 的架构设计受到现代游戏引擎设计原则的启发。尽管 3D 游戏的要求比其他视觉应用更高，但它们有大量的共同点。有些非游戏应用需要游戏引擎中的许多功能，例如，教育仿真或许需要碰撞、物理及化身，尽管它并不需要快速运行，也没人会被炸飞。

Vizi 架构的一个主要特色是*基于组件的对象模型*（component-based object model）。这反映了现代趋势，即从传统的基于继承的设计转向了组件的集合。它有一个基础对象类型 `Vizi.Object`，而不仅是组件的容器。组件实现了大部分的功能，比如一个 `Visual` 组件实现了几何体和材质、一个 `Picker` 组件实现了基于每个对象的鼠标事件派发、一个 `Camera` 组件实现了视图。基于组件的系统提供了获得能力的统一模型，它允许非常灵活的实现以及高可重用性。它同样是支持扩展的关键。

Vizi 架构还有以下亮点。

- *应用对象*

  单例应用对象（singleton application object）负责 WebGL 上下文的创建、DOM 事件处理器以及 Three.js 的初始化。应用对象实现了运行循环；对象只需要加入到应用中，就能在每帧动画更新自己。

- *模拟及事件模型*

  Vizi 有一个所有对象使用的标准时间轴。事件在定义好的时间点上触发，并遵循预定的规则。一个对象可以发布其他对象订阅的事件。对象可以使用监听器来订阅事件，或者在行为链上直接连接到其他对象的事件上。这让创建行为和交互变得非常简洁。

- *服务化架构*

  所有子系统都以黑盒服务的方式构建。在初始化和执行阶段，应用将时间、事件、图形及输入等委托给了各个服务，而不关心这些服务具体是怎么做的。这让添加新的服务变得简单，比如并没有包含在核心版本中的多用户网络。

- *图形*

  所有图形都是使用 Three.js 绘制的。Vizi 并没有将 Three.js 隐藏起来，而是直接使用它，将 Three.js 对象通过组件化的结构封装起来，使得其他 Vizi 对象可以方便地与它们进行通信。

- *交互*

  Vizi 支持基于每个对象的鼠标事件，底层使用 Three.js 的 `Projector` 类来实现点击检测。这使得鼠标和触摸输入的接口变得简单。Vizi 还提供了预建交互对象来实现不同类型的拖拽（比如在平面或球体上）。

- *行为*

  Vizi 包含了多种预建行为，能自动旋转、移动、弹跳、高亮等来改变对象的状态。

- *高级视图模型*

  Vizi 允许定义多个相机，可以轻松在它们之间切换。Vizi 同时支持多种导航模式，比如物体查看、第一人称游戏、建筑浏览等。

- 方便自定义

  自定义组件可以实现全新的行为、交互和相机控制等几乎所有事情。组件继承自 `Vizi.Component` 的 JavaScript 对象。开发者很容易就能创建一个新的组件类型，并通过覆盖 `realize()` 和 `update()` 方法来增加自定义功能。组件可以无侵入地添加到已存在的对象中，来增加原开发者没想到的新功能。

- 预制结构

  Vizi 允许开发者创建由一组对象组成的可重用类型。因为 Vizi 是基于组件设计的，所以类型并不是使用 JavaScript 的类来创建的，而是一堆被称为 prefab 的预制对象。prefab 由层次化的游戏对象及它们的组件、一个或多个事件订阅或连接，以及一个控制器脚本（协调所有部分的交互）构成。

> Vizi 基于组件的架构设计受到一本书的极大影响，这本书是 Jason Gregory 的教科书《游戏引擎架构》（*Game Engine Architecture*，http://www.gameenginebook.com/），这是高级引擎和框架设计师必读之书。这本书涉及的范围很广，但与本书最相关的是 Gregory 对对象模型架构的探索。他大力提倡组件化的设计，而不是传统的类继承设计。基于组件的设计一般更加灵活和可扩展，避免了基于继承设计遇到的许多已知问题，尤其是当复杂度上升的时候。
>
> Vizi 同样受到 Unity（http://unity3d.com/）设计的启发，Unity 是今天在独立开发者和小工作组中最流行的商业游戏引擎。Unity 很好地体现了 Gregory 基于组件化引擎设计的思想。在本书编写之际 Unity 还不支持 WebGL[2]。Unity 早在 HTML5 流行前就已开发出来了，所以它使用了它自己的脚本语言和渲染系统。如果 Unity 支持 HTML5 和 WebGL，我或许会觉得没必要创建 Vizi。

9.3.3 Vizi入门

为了开始学习 Vizi，我们先要从 GitHub（https://github.com/tparisi/Vizi）项目中获得最新版本的代码。在 engine/build/ 目录下，你会看见几个文件，复制一个 vizi.js（未压缩的 debug 版本）文件或 vizi.min.js（压缩后的 release 版本）到你的项目中。

现在，只需要将 Vizi 脚本插入你的页面中就可以开始使用了：

```
<script src="../<path_to_vizi>/vizi.js"></script>
```

> Vizi 有多种不同类型的版本。这两个文件将所有依赖都打包进来了，包括 Three.js、Tween.js、RequestAnimationFrame.js 以及几个支持 Three.js 的对象。如果你不想要包含外部依赖的版本，可以使用无依赖版本，在你页面的其他地方自行插入依赖。当然，如果不小心可能会有版本冲突。请查看 README 文件及版本公告来了解更多详情，以及查阅参考附录来了解准备自定义版本 Vizi 的更多信息。

注 2：Unity 5 已经支持。——译者注

9.3.4 一个Vizi应用示例

让我们来看一个具体的例子，通过它来了解 Vizi 框架的强大。在你的浏览器中打开示例文件 Chapter 9/vizicube.html。你应该会看见熟悉的东西，它是第 2 章和第 3 章的带纹理立方体，基于 Vizi 重写了一遍。例 9-4 是基于 Vizi 的代码，可以和第 3 章中例 3-1 的代码做比较。

例 9-4：一个 Vizi 应用的例子——旋转的立方体

```
<script type="text/javascript">

    $(document).ready(function() {

        // 创建Vizi应用对象
        var container = document.getElementById("container");
        var app = new Vizi.Application({ container : container });

        // 创建一个以phong着色法渲染的纹理映射立方体
        var cube = new Vizi.Object;
        var visual = new Vizi.Visual(
                { geometry: new THREE.CubeGeometry(2, 2, 2),
                    material: new THREE.MeshPhongMaterial(
                      {map:THREE.ImageUtils.loadTexture(
                         "../images/webgl-logo-256.jpg")})
                });
        cube.addComponent(visual);

        // 添加一个旋转行为,使得立方体更加生动
        var rotator = new Vizi.RotateBehavior({autoStart:true});
        cube.addComponent(rotator);

        // 将立方体向着观察者旋转,以便更好地展示3D细节
        cube.transform.rotation.x = Math.PI / 5;

        // 添加一个灯光来展示渲染效果
        var light = new Vizi.Object;
        light.addComponent(new Vizi.DirectionalLight);

        // 将立方体和灯光添加到场景中
        app.addObject(cube);
        app.addObject(light);

        // 运行
        app.run();
    }
    );

</script>
```

基于 Vizi，只用了大概 40 行代码就创建了一个旋转的带纹理立方体，而我们之前使用 Three.js 则需要写 80 行代码。但我们稍后会看到代码量少只是其中的一个优点。我们来通读这个例子。首先我们创建了一个新的应用对象，类型是 `Vizi.Application`，并将容器元素传递给它。这个操作在内部做了大量工作：创建一个 Three.js 渲染对象和一个带默认相机的空 Three.js 场景，增加了监听页面缩放、鼠标及其他 DOM 事件的事件处理程序。这

些事情你本来需要通过 DOM API 或调用 Three.js 函数来手工操作，但 Vizi 自动进行了处理。查看 Vizi 源码中的文件 core/application.js 和 graphics/graphicsThreeJS.js 来了解具体的实现细节，它需要做许多事情。

接下来，我们将对象加入到场景中。这时轮到 Vizi 组件对象模型出场了。Vizi 场景中的任意对象都是 `Vizi.Object` 类型的，然后我们添加不同的组件给它。对于立方体来说，我们使用 Three.js 的立方体几何体和带纹理的 Phong 材质创建了一个 `Vizi.Visual` 对象。注意 Vizi 并没有定义自己的图形对象，而是选择使用 Three.js 来实现所有图形。这是有意的选择。我们完全暴露了 Three.js 的强大功能，而不是试图隐藏它，因此我们可以轻松创建我们需要的任何视觉类型。

在创建可视化组件并添加到对象上后，我们添加了一个行为。这是 Vizi 神奇的地方。Vizi 自带了一堆预制的行为，我们可以通过添加组件的方式应用到对象上。在这个例子中，我们添加了一个 `Vizi.RotateBehavior` 行为，设置 `autoStart` 标记为 `true`，使得对象在应用运行的时候马上开始旋转。

我们想让立方体向相机倾斜，以便于查看 3D 效果。在 Vizi 中，我们可以通过修改对象变换组件的 `rotation` 属性来实现。

```
// 将立方体向着观察者旋转,以便更好地展示3D细节
    cube.transform.rotation.x = Math.PI / 5;
```

注意，为了方便，默认为每个 Vizi 对象都自动创建了一个变换组件。这覆盖了大部分的使用场景。`Vizi.Object` 的构造器还有一个可选参数 `autoCreateTransform`，如果某个对象不需要变换组件，可以将它设置为 `false`。

为了显示 Phong 着色器的效果，我们通过另一个对象来添加光源，这个对象中有 `Vizi.DirectionalLight` 组件。在后面几章中，我们会看到如何避免显式创建光源，那就是使用带灯光设置的预制应用模板。最后，我们准备好运行应用了，于是调用应用的 `run()` 方法。这就够了，不需要编写我们自己的 `requestAnimationFrame()` 函数来手动在每个周期更新立方体的旋转。

1. 添加交互

你或许注意到前面几章介绍的 Three.js 例子并没有交互性，部分原因是我们还没有介绍到交互，另一个原因是使用 Three.js 创建交互涉及枯燥的工作。Three.js 提供了一个 projector 对象，允许我们计算出当前鼠标悬停在哪个对象上。但它并没有封装成一个事件接口或者一个点击拖拽的模型。Vizi 框架通过组件来实现自动化鼠标选择和事件分发，以此来解决这个问题。

让我们添加一个简单的交互行为到前面的例子中。我们将只在鼠标悬停上去的时候旋转立方体，而不是页面加载后。在浏览器中打开文件 Chapter 9/vizicubeinteractive.html，代码如例 9-5 所示。粗体标注的代码行显示了修改了的部分。这一次我们不在创建行为的时候设置 `autoRotate` 选项，因此它不会在应用加载的时候就开始旋转。接下来，我们添加一个新的组件类型 `Vizi.Picker` 到立方体对象上。picker 定义了通常的鼠标事件，包括 over（移到元素上）、out（移出元素外）、up（在元素上松开鼠标按钮）和 down（在元素上按下鼠标按钮），它们会在鼠标移动到包含 picker 的对象时触发。只需要分别在鼠标悬停和移出

的时候添加事件监听，来开始和停止旋转。

例 9-5：使用一个 picker 组件来添加鼠标交互

```
<script type="text/javascript">

   $(document).ready(function() {

       // 创建Vizi应用对象
       var container = document.getElementById("container");
       var app = new Vizi.Application({ container : container });

       // 创建一个以phong着色法渲染的纹理映射立方体
       var cube = new Vizi.Object;
       var visual = new Vizi.Visual(
               { geometry: new THREE.CubeGeometry(2, 2, 2),
                   material: new THREE.MeshPhongMaterial(
                     {map:THREE.ImageUtils.loadTexture(
                        "../images/webgl-logo-256.jpg")})
               });

       cube.addComponent(visual);

       // 添加一个旋转行为,使得立方体更加生动
       var rotator = new Vizi.RotateBehavior;
       cube.addComponent(rotator);

       // 使立方体可以被选中
       var picker = new Vizi.Picker;
       cube.addComponent(picker);

       // 将选择器和旋转器关联起来,仅在鼠标悬停时旋转
       picker.addEventListener("mouseover", function() {
         rotator.start(); });
       picker.addEventListener("mouseout", function() {
         rotator.stop(); });

       // 将立方体向着观察者旋转,以便更好地展示3D细节
       cube.transform.rotation.x = Math.PI / 5;

       //  添加一个灯光来展示渲染效果
       var light = new Vizi.Object;
       light.addComponent(new Vizi.DirectionalLight);

       // 将立方体和灯光添加到场景中
       app.addObject(cube);
       app.addObject(light);

       // 运行
       app.run();
   }
   );

</script>
```

这非常简单。为了研究底层究竟发生了什么事情，让我们看看检测到 3D 对象在鼠标下时

发生了什么。这 Vizi 基于 Three.js 的 THREE.Projector 类来实现选取功能的方式。它并不简单。例 9-6 列出了 Vizi 图形子系统的 objectFromMouse() 方法的代码。这个方法返回鼠标指针下能找到的对象，它的处理包括以下几步。

(1) 首先，我们将鼠标事件相对于元素的坐标 elementX 和 elementY 转换成相对于视口的坐标，每个维度的值的范围都是 −0.5 到 +0.5，同时翻转 y 坐标来匹配 3D 坐标系统。（注意 elementX 和 elementY 并不是标准的 DOM 鼠标事件属性，它们在参数传递之前，会由 Vizi 的 DOM 事件处理函数计算好。）

(2) 当获得了鼠标相对视口的坐标后，需要将它们转换到鼠标下面的 3D 位置，并保存在变量 vector 中。

(3) 然后需要将鼠标相对于视口的位置从*相机空间*转换到*世界空间*中。一旦转换了位置，我们就知道鼠标指针在世界空间中的位置了。为此，我们调用投影对象的 unprojectVector() 方法。（投影对象是在初始化图形系统的时候创建的，它是 THREE.Projector 类型。）

(4) 现在鼠标指针在场景中的位置已经计算出来了，我们可以创建一束光线，从相机的世界空间位置到鼠标的世界空间位置。任何和光线相交的对象都是“在鼠标下”的。光线相交是通过 THREE.Raycaster 的 intersectObjects() 方法来实现的。向它传递对象列表，它会按照从前到后的顺序返回任何与光线相交的对象列表。

(5) 最后，我们取得列表中第一个可见的元素，它就代表了最前面的被选中的对象。

我说过，这不简单。这样的代码你可不想写多次。而使用 Vizi 这样的框架，你完全不需要自己动手。

例 9-6：使用 THREE.Projector 实现 Vizi 选取功能

```
Vizi.GraphicsThreeJS.prototype.objectFromMouse = function(event)
{
    var eltx = event.elementX, elty = event.elementY;

    // 将客户端(视口)坐标转换为世界坐标
    var vpx = ( eltx / this.container.offsetWidth ) * 2 - 1;
    var vpy = - ( elty / this.container.offsetHeight ) * 2 + 1;

    var vector = new THREE.Vector3( vpx, vpy, 0.5 );

    this.projector.unprojectVector( vector, this.camera );

    var pos = new THREE.Vector3;
    pos = pos.applyMatrix4(this.camera.matrixWorld);

    var raycaster = new THREE.Raycaster( pos, vector.sub( pos )
      .normalize() );

    var intersects = raycaster.intersectObjects( this.scene.children,
      true );

    if ( intersects.length > 0 ) {
        var i = 0;
        while(!intersects[i].object.visible)
        {
            i++;
```

```
        }

        var intersected = intersects[i];

        if (i >= intersects.length)
        {
            return { object : null, point : null, normal : null };
        }

        return (this.findObjectFromIntersected(intersected.object,
            intersected.point, intersected.face.normal));
    }
    else
    {
        return { object : null, point : null, normal : null };
    }
}
```

2. 添加多个行为

Vizi 允许我们轻松地给一个对象添加多个行为。我们将修改前面的例子，给立方体添加多个行为。当鼠标悬停的时候，它会变成浅蓝色来高亮；如果点击鼠标，它会开始旋转、跳上跳下、慢慢远离镜头；再次点击立方体它会停止移动。打开文件 Chapter 9/vizicubebehaviors.html 来查看效果。相关代码在例 9-7 中，粗体标注的代码行显示了修改了的部分。

例 9-7：添加多个行为

```
         // 添加一个复合行为
        var rotator = new Vizi.RotateBehavior;
        var bouncer = new Vizi.BounceBehavior({loop:true});
        var mover = new Vizi.MoveBehavior({loop:true, duration:2,
          moveVector:new THREE.Vector3(0, 0, -2)});
        cube.addComponent(rotator);
        cube.addComponent(bouncer);
        cube.addComponent(mover);

        // 使立方体可以被选中
        var picker = new Vizi.Picker;
        cube.addComponent(picker);

        // 添加悬停高亮颜色
        var highlight = new Vizi.HighlightBehavior(
          {highlightColor:0x88eeff});
        cube.addComponent(highlight);

        // 将选择器和旋转器关联起来。鼠标悬停时高亮,点击时切换行为
        picker.addEventListener("mouseover", function() {
          highlight.on(); });
        picker.addEventListener("mouseout", function() {
          highlight.off();});
        picker.addEventListener("mouseup", function() {
              rotator.toggle();
               bouncer.toggle();
               mover.toggle(); });
```

在这个例子中添加行为就像添加更多组件一样容易。我们还添加了鼠标松开的事件监听，它会调用每个行为的 `toggle()` 方法，来在点击鼠标的时候触发各个行为的开始 / 停止状态。图 9-5 显示了我们的老朋友，带纹理的立方体——高亮、旋转、摆动、远离，直到消失在远处。



图 9-5：Vizi 的带纹理立方体，具有行为和鼠标交互

这些简单的例子展示了 Vizi 这样的框架能做什么。我们会在后面几章中进一步了解它，还会介绍各种 3D 应用的方案和技术。

> Vizi 还在开发之中。本书编写的时候，它是 0.6 或 0.7 版本，有很多功能，但还有很长的路要走。我在空闲或有机会开发 WebGL 项目的时候开发它。当这本书付印的时候，我希望已经发布 1.0 版本了。

9.4　小结

本章介绍了构建 3D 应用的引擎和框架。尽管 Three.js 很强大，但它缺乏让日常开发变得可管理的功能，比如高层的行为和交互、预制的启动和关闭，以及许多几乎所有 3D 应用中都必须完成的事情。

我们还介绍了解决这些问题的现有游戏引擎和展示框架。WebGL 开发这一领域还很年轻，在不断地发展，目前的框架也是如此。有很多可选的框架，需要作出不少权衡，包括控制权、功能、易用性，等等。

最后，我们使用 Vizi 来深入研究了基于框架的开发，它是我开发的新框架，用于探索框架

概念，以及开发本书的例子。简单的 Vizi 例子展示了使用框架为何能将我们从普遍的、重复的开发任务中解放出来，帮助我们节省时间，编写更可靠的代码，并专注于开发本身。Vizi 只是框架的一个例子，你还需要探索其他的框架，甚至是设计你自己的框架。无论是选择购买还是自行开发，如果你要开发产品级别的 3D 应用，很快就需要解决本章讨论过的那些问题。

第10章

开发一个简单的3D应用

目前为止，我们关注的都是基础：HTML5 基础 API 和架构、JavaScript 库和框架，以及内容制作流程的工具。现在是时候实践这些知识了。在本书剩下的部分，我们将不再关注 API 和工具，而是转向开发实际应用的具体实践。

我们从创建一个简单类型的 3D Web 应用开始：一个产品的查看器 / 配置器。这样的应用通常会是一个真实产品的交互式 3D 模型，有丰富的界面供探索产品的功能，并可以使用鼠标交互来查看更多信息，以及提供某些方法来改变模型的一个或多个属性。基于 Web 的产品配置工具已经出现了很长时间：首先是静态的 2D 模式，然后是使用 Flash 的 2.5D 或 3D 模式，最近出现了基于 Canvas API 或 WebGL 的 3D 模式。产品配置器既可以是高功能营销工具（即一种交互性更强的，为产品功能做广告的方式），也可以集成到电子商务系统中，用来真实地在线配置和购买产品。

图 10-1 展示了一个"未来车"的概念设计。可以打开文件 Chapter 10/futurgo.html 来体验它。用鼠标来旋转模型，用触摸板或滚轮来放大和缩小。当鼠标悬停在车上不同部分的时候，关于这部分的信息就在一个浮层上显示出来。点击右边的标签来显示车的内部结构以及转动轮子。你甚至可以根据你的个人喜好来改变车的颜色。"生活方式型交通工具"是个人交通工具的下一波浪潮。由小摩托车、高尔夫球车、智能车的各一部分，以及满满的高科技组合而成，这就是 Futurgo！

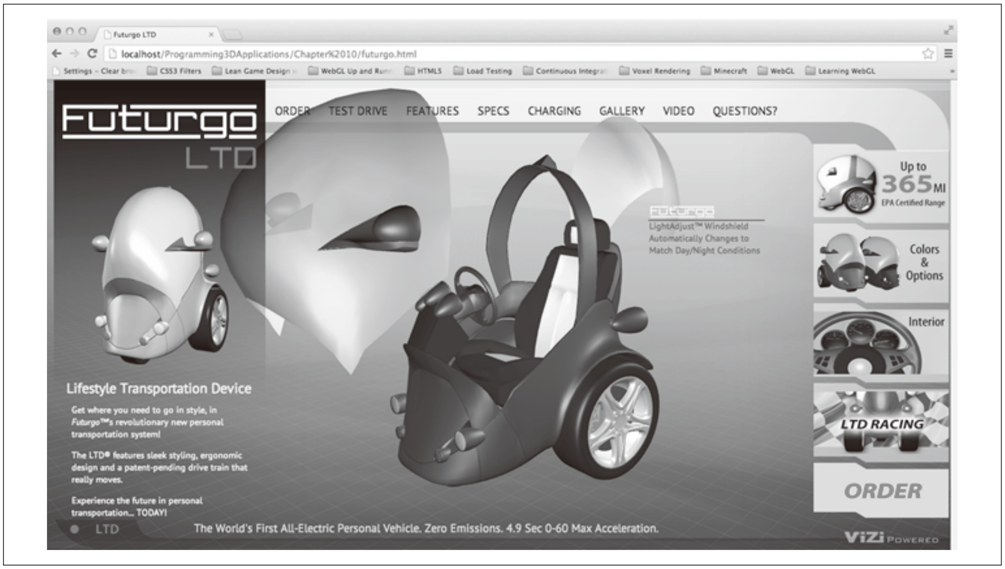


图 10-1：Futurgo 概念车——一个 3D 产品页面

这是一个有趣的、完全人造的产品配置器例子，它涉及了部署一个 3D 产品页面所需的关键概念。

- 设计应用

  开发 3D 模型及 2D 页面内容的视觉效果，并定义用户交互流程。

- 创建 3D 内容

  使用像 Autodesk Maya 这样的工具来创建模型和动画，并转换为适合 Web 的格式来在应用中使用。

- 预览和测试 3D 内容

  设计一系列的工具来验证导出的 3D 内容可以在应用中工作（如外观和动画正常）。

- 集成 3D 内容到应用中

  将 3D 内容（一旦验证它的外观和动画正常）与 2D 页面内容以及其他应用代码集成。

- 开发 3D 行为和交互

  通过实现一些行为和交互来让 3D 内容生动起来，包括一个视觉淡出效果、一个旋转台、鼠标移入效果、用户交互触发的动画，以及动态修改对象的颜色。

上述所有需求需要通过一个可重复的流程来开发。当我们排除 bug 和改善应用后，应该达到可以随意修改修改应用的可视部分（尤其是 3D 内容），而不需要重新编写应用代码的状态。

为了构建 Futurgo 的页面，我们需要工具来支持它。我们会深入学习第 9 章介绍过的 Vizi 框架。Vizi 基于 Three.js 开发，提供了可重用的行为及封装好的对象，让我们的工作变得简单，并允许我们用更少的代码做更多的事情。我们还会使用开源文件导出工具和转换工具，来将内容从 Maya 中导出成适合 Web 的格式。让我们现在就开始吧。

10.1 设计应用

我和 3D 设计师 TC Change 一起设计了 Futurgo 应用。我和 TC 希望将 Futurgo 设计为产品可视化应用，它同时具备可玩性和未来主义风格。Futurgo 概念车设计融合了类似 Segway 这样的个人交通工具的特点，同时还具备汽车的防护性，比如封闭式车身和挡风玻璃。我们不知道这台车是否真的能用，更别说合法上路了，但我们觉得将这些概念放到一起很有趣。

经过基本的概念讨论及草图设计之后，TC 开始着手进行完整概念的视觉处理。原型如图 10-2 所示。注意，原型和最终产品非常接近。[1] 我们可以重用 Photoshop 的资源，将其导出为 PNG 图片，并且 3D 渲染是直接从 Maya 得到的，所以它和使用 Three.js 实时渲染的版本非常像。我们稍后会看到，为了保证导出的 3D 内容忠于原先的渲染，我们还需要做一些额外的工作。最终结果显示这是值得的。本章旨在教你如何达到这样的视觉保真度并且无缝地将艺术和代码融合起来，从而创建精良、专业的应用。



图 10-2：Futurgo 概念车的设计师原型；由 TC Change（http://www.tcchang.com/）设计并提供图片

> TC Change 是资深的美术总监，拥有傲人的简历，包括长期在迪士尼互动、索尼、艺电等公司工作，参与过诸多打斗游戏的开发，如《教父》《詹姆斯·邦德》《李连杰》。TC 坚信 Web 3D 的力量，他创建了该领域的一个早期创业公司 Flatland。你可以访问 http://www.tcchang.com/ 来在线查看 TC 的工作。

注 1：唯一明显的视觉差异是字体，TC 选择了一种叫作 Mayraid Pro 的字体，但它不是 Web 字体。Google Fonts 的 PT Sans（http://www.google.com/fonts/specimen/PT+Scans）是一个不错的替代。

10.2　创建3D内容

TC 使用 Autodesk Maya 2013 版来创建 Futurgo 模型的各种部分并添加动画。虽然 Futurgo 在概念上是一个物体，但它实际上是由多个网格组成的，每个代表车的一部分：钢质车身、车轮、车内座位及操控、车窗等。使用单独的各个部分来创建模型是很重要的，这样每个部分就能单独进行动画，从而可以实现应用中的不同交互，比如鼠标悬停在车窗或车身骨架上来获得更多相关信息。图 10-3 展示了在 Maya 中对 Futurgo 进行建模。覆盖在上面的文本显示了关于模型的各种统计信息，包括顶点和三角形的数量。

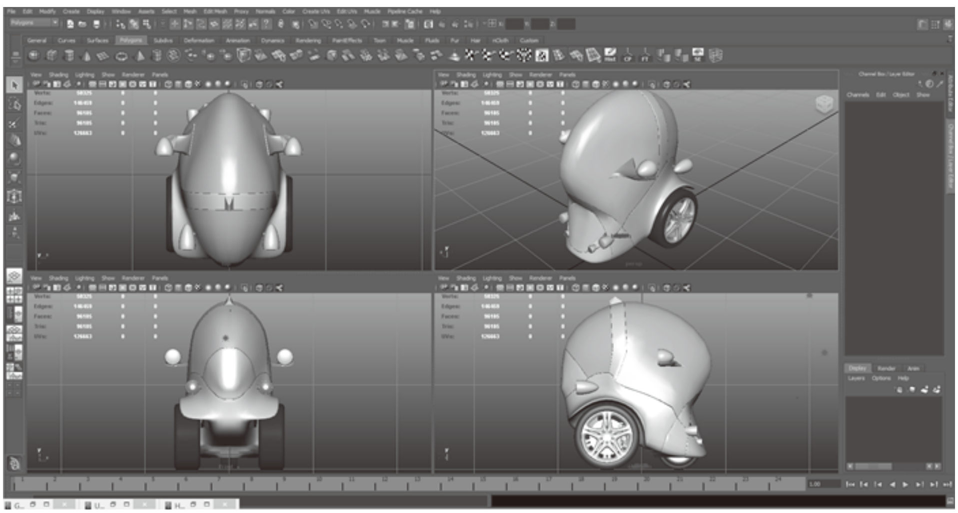


图 10-3：在 Autodesk Maya 中对 Futurgo 进行建模；图片由 TC Change 提供（http://www.tcchang.com/）

在内容创建这一阶段，我和 TC 仔细规划了模型的各个方面，如比例（Futurgo 模型的度量单位，这里是米）、设置光源的方式、创建动画的方式。Maya 在动画工具方面有所局限，整个文件只能有一条动画时间线，因此文件中的所有动画都需要相同的时长，不然短的间隔就会出现空白动画。我们决定保持动画简短，时长只有一秒，这足够将车窗从车身上分离出来以显示内饰，也足够完成一个轮子的全程旋转了。

为保证性能，我和 TC 还控制了多边形的数量。Futurgo 使用了约 96 000 个三角形，这就保证了既有足够的三角形来在 Three.js 中渲染出平滑的效果，也不至于让浏览器的性能陷入困境。这样做的另一个目的在于我们需要通过控制多边形的数量来限制文件体积。对于 Web 应用来说，内容需要快速下载。最终我们从 Maya 转成 glTF 的部署文件大概 6 MB。这看起来还是有点大，但对于现代用户的网络来说，从配置了服务器端 .bin 文件压缩的服务器将这些内容流式下载到本地，速度还是非常快的（只需几秒钟）。

10.2.1 导出Maya场景到COLLADA

在供浏览器展示前，Maya 文件需要先转成适合 WebGL 的格式。我们决定使用 glTF 作为部署格式，因为它体积小且加载速度非常快。在本书编写的时候，还没有直接从 Maya 导出为 glTF 的方法，因此我们使用 Maya 支持导出的格式 COLLADA，然后再转成 glTF。[1]

Maya 2013 的 COLLADA 导出工具有很多问题而且过时了，所以我们选择使用 OpenCOLLADA（http://opencollada.org/），一个高性能的开源导出工具，由独立开发者开发，可以创建高质量、兼容规范的 COLLADA 输出。本书编写的时候，OpenCOLLADA 的 Maya 版本（还有 3ds Max 版本）工作良好，我们可以成功使用它将 Futurgo 导出为 COLLADA 格式。OpenCOLLADA 的主站包含 Maya 插件及 3ds Max 从 2010 到 2013 版本插件的下载（Autodesk 倾向于每年更新它的插件 SDK，因此导出工具需要跟上，以保证导出工具的版本匹配产品版本）。导出工具安装好后，需要保证它在 Maya 中处在启用状态，具体配置在插件管理器中（窗口→设置 / 首选项→插件管理器），如图 10-4 所示。

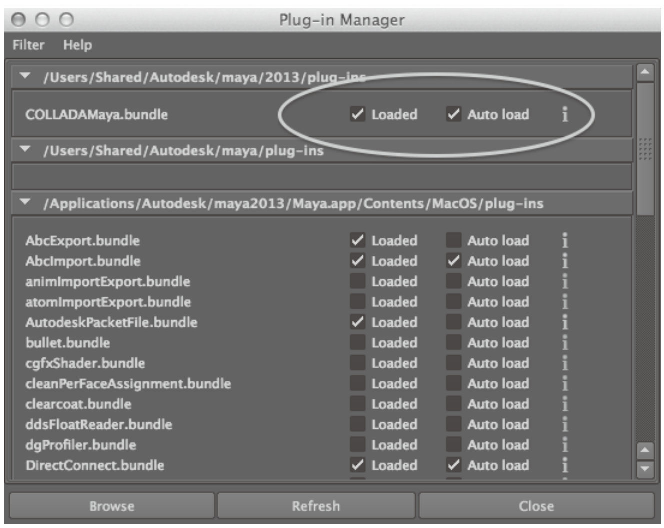


图 10-4：在 Autodesk Maya 2013 插件管理器中启用 OpenCOLLADA 导出

Futurgo 导出的 COLLADA 文件和示例代码在一起，文件是 models/futurgo/futurgo.dae。

> OpenCOLLADA 是一个开源项目，它的更新和维护有点类似于义务劳动的性质，所以使用的时候需要注意。在这里，我们并不能保证它会一直维护下去，尤其是在 Autodesk 后续版本中更新 SDK 的时候。不过 COLLADA 只是导出 Maya 和 3ds Max 文件的一种方法，另外一种有可能的方法是将 FBX 转成 glTF。Autodesk 的工具在今后一段时间能保证输出 FBX 格式。一些公司，比如第 8 章介绍的 Verold（http://www.verold.com/），正在尝试将 FBX 转成 glTF。

注 1：glTF, 用于 WebGL 的图形库传输格式；COLLADA，基于 XML 的图形交换格式。具体内容参见第 8 章。

10.2.2　将COLLADA文件转换glTF格式

当 Futurgo 模型从 Maya 导出为 COLLADA 后，它就可以被转成 glTF 格式了。COLLADA 工作组的主席及 glTF 的主设计师 Fabrice Robinet 写了一个命令行工具来进行这个工作。

在我的 MacBook Air 运行的 Mac OS 10.8 上，执行命令的是一个叫 collada2gltf 的可执行文件。要转换 Futurgo，我运行如下命令，程序的输出使用斜体表示。

```
$ <path-to-converter>/collada2gltf -f futurgo.dae -d

[option] export pass details
converting:futurgo.dae ... as futurgo.json
[shader]: futurgo0VS.glsl
[shader]: futurgo0FS.glsl
[shader]: futurgo2VS.glsl
[shader]: futurgo2FS.glsl
[shader]: futurgo4VS.glsl
[shader]: futurgo4FS.glsl
[completed conversion]
```

转换后，你会在你的目录下看到 futurgo.json 文件，以及相应的 GLSL 着色器源文件（后缀是 .glsl）。现在文件被转成了 glTF 格式，可以使用我为 Three.js 编写的 glTF 加载器来将其加载到应用中。我们会在下个小节介绍如何做到这一点。Futurgo 转换后的 glTF 文件与示例代码打包在了一起，models/futurgo/futurgo.json 是主 JSON 文件，models/futurgo/futurgo.bin 是它关联的二进制文件。

本书编写的时候，glTF 还处在发展的早期阶段。这句话有两层意思。首先，规范本身还在变化，因此本书所带的文件有可能在规范明确的时候过时了，所以你需要准备更新或迁移你的内容。其次，这个工具还很年轻，比如 collada2gltf 转换工具必须在目标平台上从代码进行编译。更多关于转换工具的信息，可以访问 glTF 在 GitHub 上的项目（https://github.com/KhronosGroup/glTF）或 glTF 主页（http://gltf.gl/）。

10.3　预览和测试3D内容

现在我们已经将内容从 Maya 中导出了，接下来我们要解决下一个问题：如何让它在页面上显示出来。glTF 文件并不能直接查看——回想一下，WebGL 并不能直接识别任何格式。我们需要使用我们自己的库来加载模型和场景。在构建应用前，最好能先保证 3D 内容是良好的，也就是说，我们可以完整地在 WebGL 中渲染全部场景信息，比如材质、纹理、光源、相机、变换、动画。为此，我们需要创建一个工具来帮助我们预览和测试我们的 3D 内容。

10.3.1　基于Vizi的预览工具

我们使用 Vizi 来创建 3D 预览工具。我们在第 9 章介绍过 Vizi，它是我开发的框架。Vizi

使用基于组件的方式来构建 3D 应用，它可以自动完成类似初始化和清空画布这样的重复性工作，并提供了应用程序的运行循环和事件处理机制，以及一系列预建的行为和交互。虽说图形仍然使用的是 Three.js 这个 WebGL 事实上的标准库，但 Vizi 将它封装得更加可重用和可快速编码。

图 10-5 展示了预览工具载入 Futurgo 的 glTF 文件到场景中的效果。预览工具有一个主内容区域用于查看模型和场景，并带有网格地板。有一个菜单栏，其中包含一个 Open 按钮，以及显示当前浏览文件的标签。右侧是一个控制面板，带有几个子面板。场景状态（Scene Stats）面板显示了当前的帧率、网格和多边形的数量，以及加载场景所花费的时间。相机（Cameras）、光源（Lights）和动画（Animations）面板允许用户测试这些场景部件，可以切换相机、打开和关闭光源、运行动画。还有一个其他参数（Miscellaneous）面板允许我们打开和关闭前光源（它是一个额外的光源，用于在模型没有光源的时候使用），还有一个复选框来显示和隐藏网格。

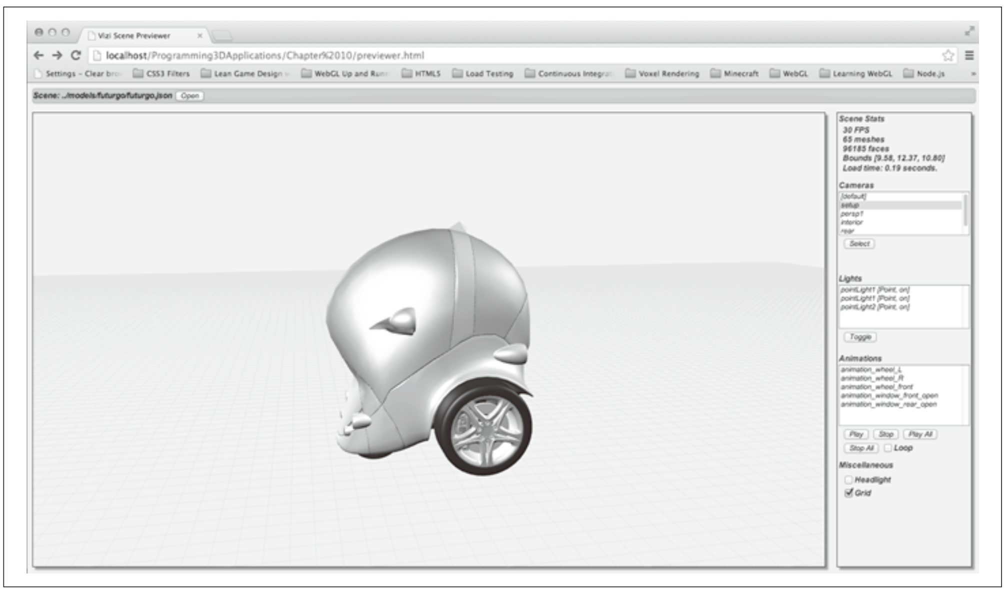


图 10-5：使用 Vizi 预览一个 glTF 模型

除了让我们可视化地查看和测试内容，预览器还提供了关于场景的重要信息，即在应用中连接交互所使用对象的 id。TC 在上传导出的 COLLADA 文件后，发邮件告诉了我动画的名称，但那并不是一个确定在代码中所使用的对象 id 的可靠方法。使用预览工具，我们可以确定在 COLLADA 和转换后的 glTF 文件中，动画所使用的对象的 id（它就是在右边控制面板中的动画面板内所显示的名称）。

本章中展示的预览器是一个简单的独立工具。虽然它不是一个完整的开发环境，但它为整个开发流程提供了在将内容添加到应用中之前必要的验证和测试环节。我们将在后续的项目开发中使用这样的预览器。现在先让我们看看它是如何创建的。

10.3.2 Vizi查看器类

许多 3D 应用都遵循同一个模式：初始化渲染器，创建一个空的场景，加载一些模型内容，添加交互，运行它。这个模式非常常见，因此我设置了一个可重用的 Vizi 类来实现它。`Vizi.Viewer` 是 `Vizi.Application` 的子类，它是一个专门用于查看模型和场景并与之交互的应用。你可以使用它来支持用鼠标旋转模型及场景，将模型及场景绘制在左边、右边、上面及下面，以及对它们进行缩放。

`Vizi.Viewer` 可以用在许多类型的浏览场景：一个简单的查看器，载入模型并允许用户查看和旋转它，没有别的附加修饰物；一个本节会介绍的预览器；甚至是一个完整的产品可视化页面，比如 Futurgo（我们会在本章后面看到它是如何做到的）。

`Vizi.Viewer` 是一个真正的框架样式的类，它在一个类中包含了许多开箱即用的功能，而自行去实现这些功能往往需要数百行 Three.js 代码。`Vizi.Viewer` 的功能如下。

- *模型查看控制*

  `Vizi.Viewer` 使用了一个来自 Three.js 例子的增强版本的 `THREE.OrbitControls`。鼠标左键旋转场景，鼠标右键平移场景，鼠标滚轮和触摸板缩放场景。它还可以覆盖和重新映射鼠标绑定。

- *默认相机和光照*

  对于未包含相机的场景，`Vizi.Viewer` 提供了一个默认的相机。对于没有光照的场景，它可以视需要创建一个前置光源来自动照亮模型，并且在用户移动相机的时候更新光源。

- *场景工具对象*

  如果需要，`Vizi.Viewer` 可以显示一个矩形网格的地面，还可以在模型周围显示边界框。

- *场景和渲染统计*

  `Vizi.Viewer` 对象可以通过发布事件来报告帧率、场景统计（如网格和多边形数量）、边界框尺寸，以及文件加载时间。

- *光源、相机和动画控制*

  `Vizi.Viewer` 提供了辅助方法来允许程序员打开和关闭光源、切换相机、开始和停止动画。它还为应用提供了每个类型的对象的列表，因此不需要在代码中查找它们。

- *一键操作*

  `THREE.OrbitControls` 对象被修改为支持一键操作，因此为了可用性，鼠标右键会被禁用或映射为鼠标左键。

为了查看 `Vizi.Viewer` 的实际效果，让我们看看它是如何用来实现预览工具的。打开文件 Chapter 10/previewer.html 来查看图 10-5 所示的预览器效果。例 10-1 展示了创建 `Vizi.Viewer` 对象的部分源代码。和之前一样，我们传递一个 `container` 参数，它是一个 DIV 元素，Three.js 会将它的 WebGL 渲染器（也就是一个带 WebGL 绘图上下文的 Canvas）添加到上面。我们还增加了一些选项，来告诉查看器显示网格、使用一个前置光源（这样如果场景中没有光源我们也可以看到模型）。创建好查看器对象后，我们添加几个事件监听器，来检测帧率及其他场景状态的变化。我们稍后会研究这些事件处理函数。接下来我们为菜

单栏的 Open 按钮构建了一个文件列表。最后我们调用查看器的运行循环来运行应用。

例 10-1：创建 Vizi 查看器对象

```
   var viewer = null;
$(document).ready(function() {

    var container = document.getElementById("container");
    var renderStats = document.getElementById("render_stats");
    var sceneStats = document.getElementById("scene_stats");

    viewer = new Vizi.Viewer({ container : container,
           showGrid : true, headlight : true,
          showBoundingBox : false });
    viewer.addEventListener("renderstats", function(stats) {
       onRenderStats(stats, renderStats); });
    viewer.addEventListener("scenestats", function(stats) {
       onSceneStats(stats, sceneStats); });

    buildFileList();

    viewer.run();
}
);
```

打开预览工具，你会看到一个空白的场景窗口。顶部橙黄色的菜单栏提供一个用户界面来打开文件列表中的一个 3D 文件。

10.3.3 Vizi加载类

点击顶部的 Open 按钮会打开一个文件菜单，我们可以选择文件。选择 ../models/futurgo/futurgo.json 文件。你会看到 Futurgo 显示在场景窗口中，如图 10-5 所示。你可以随意使用鼠标、触摸板或滚轮来对其进行交互。

预览工具使用另一个 Vizi 类——`Vizi.Loader`，来将模型加载到 Vizi 查看器对象中，请看例 10-2。

例 10-2：使用 `Vizi.Loader` 对象来加载文件

```
function openFile()
{
    var select = document.getElementById("files");
    var index = select.selectedIndex;
    if (index >= 0)
    {
        var url = select.options[index].text;

        var loader = new Vizi.Loader;

        loader.addEventListener("loaded", function(data) {
          onLoadComplete(data, loadStartTime); });
        loader.addEventListener("progress", function(progress) {
          onLoadProgress(progress); });

        var fileViewingName = document.getElementById("fileViewingName");
```

```
            fileViewingName.innerHTML=url;

            var loadStartTime = Date.now();
            loader.loadScene(url);

            var loadStatus = document.getElementById("loadStatus");
            loadStatus.style.display = 'block';
        }

        $('#fileOpenDialog').dialog("close");
    }
```

Vizi.Loader 使用 Three.js 的 JSON、COLLADA 或 glTF 加载器来解析各种格式，并将它们加载到内存中。并且，它还在 Vizi 组件中封装了新创建的 Three.js 场景，使得场景适合在基于 Vizi 的应用中使用。最后，它在文件下载和解析的时候对监听器发送 loaded 和 progress 事件。例 10-3 展示了事件监听函数 onLoadComplete()，它用来检测文件何时加载完成并准备好添加到查看器中。

例 10-3：预览工具的文件加载事件监听函数

```
    function onLoadComplete(data, loadStartTime)
    {
        // 隐藏加载状态栏
        var loadStatus = document.getElementById("loadStatus");
        loadStatus.style.display = 'none';

        viewer.replaceScene(data);

        var loadTime = (Date.now() - loadStartTime) / 1000;
        var loadTimeStats = document.getElementById("load_time_stats");
        loadTimeStats.innerHTML = "Load time: " +
          loadTime.toFixed(2) + " seconds."

        updateCamerasList(viewer);
        updateLightsList(viewer);
        updateAnimationsList(viewer);
        updateMiscControls(viewer);

        if (viewer.cameraNames.length > 1) {
            selectCamera(1);
        }

    }
```

这个监听函数做了几件事情。首先，它隐藏了用于显示“Loading scene...”信息的 DIV 元素，从而告诉用户加载已经完成了。接下来，它调用查看器的 replaceScene() 方法来将新加载好的内容添加到查看器中，以允许我们在场景窗口中查看和操作 Futurgo。然后这个监听函数更新场景状态面板中的加载时间。接下来，基于查看器中被操作对象的数组，它调用几个辅助函数来更新用户界面中的列表（例如相机和光源）。最后，它调用函数 selectCamera() 来选择场景中的第一个相机（不包含默认那个），如果它存在的话。selectCamera() 使用查看器的 useCamera() 方法来切换相机：

```
function selectCamera(index)
{
    var select = document.getElementById("cameras_list");
    if (index === undefined) {
        index = select.selectedIndex;
    }
    else {
        select.selectedIndex = index;
    }

    if (index >= 0) {
        viewer.useCamera(index);
    }
}
```

现在，我们可以使用预览工具来检查和测试模型了。使用鼠标左键旋转模型，鼠标右键平移模型，鼠标滚轮或触摸板缩放模型。试试右边的各种控制来切换相机、打开和关闭灯光，以及播放动画。

能测试动画是预览工具一个很重要的功能。3D 动画很容易出错，在制作过程中会有各种可能的问题出现。在预览工具中测试动画可以节省后续解决故障所花费的时间。图 10-6 展示了我们播放动画 `animation_window_front_open` 和 `animation_window_rear_open` 的效果。可以看到我们得到了想要的效果：前面和后面的车窗弹开了，所以我们可以看到模型的内饰。

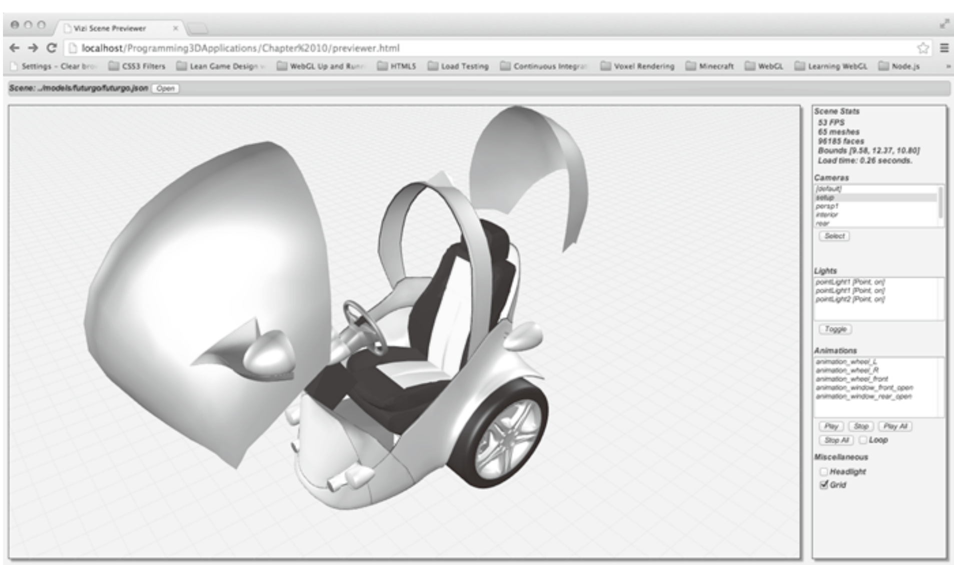


图 10-6：在预览工具中播放动画

注意动画并不是 Vizi 场景图（或者说 Three.js 场景图）的一部分。它们在查看器中保存在一个独立的对象数组中。因此我们需要使用查看器的工具方法来播放、停止和循环动画。例 10-4 展示了 HTML 中的相关函数，它们调用查看器的不同方法来播放、停

止和循环动画，包括 `viewer.playAnimation()`、`viewer.stopAnimation()`、`viewer.playAllAnimations()`、`viewer.stopAllAnimations()` 和 `viewer.setLoopAnimations()`。

例 10-4：使用 Vizi 查看器的方法来控制动画播放

```
function selectAnimation()
{
    var select = document.getElementById("animations_list");
    var index = select.selectedIndex;
    if (index >= 0)
    {
        viewer.playAnimation(index, viewer.loopAnimations);
    }
}

function playAnimation()
{
    var select = document.getElementById("animations_list");
    var index = select.selectedIndex;
    if (index >= 0)
    {
        viewer.playAnimation(index, viewer.loopAnimations);
    }
}

function stopAnimation()
{
    var select = document.getElementById("animations_list");
    var index = select.selectedIndex;
    if (index >= 0)
    {
        viewer.stopAnimation(index);
    }
}

function playAllAnimations()
{
    viewer.playAllAnimations(viewer.loopAnimations);
}

function stopAllAnimations()
{
    viewer.stopAllAnimations();
}

function onLoopChecked(elt)
{
    viewer.setLoopAnimations(elt.checked);
}
```

10.4　将3D集成到应用中

现在我们确信我们的 3D 内容会正常加载、渲染和动画，可以开始构建应用了。第一步是将 3D 集成到应用的 Web 页面中。再次打开文件 Chapter 10/futurgo.html 来查看页面，如图

10-7 所示。

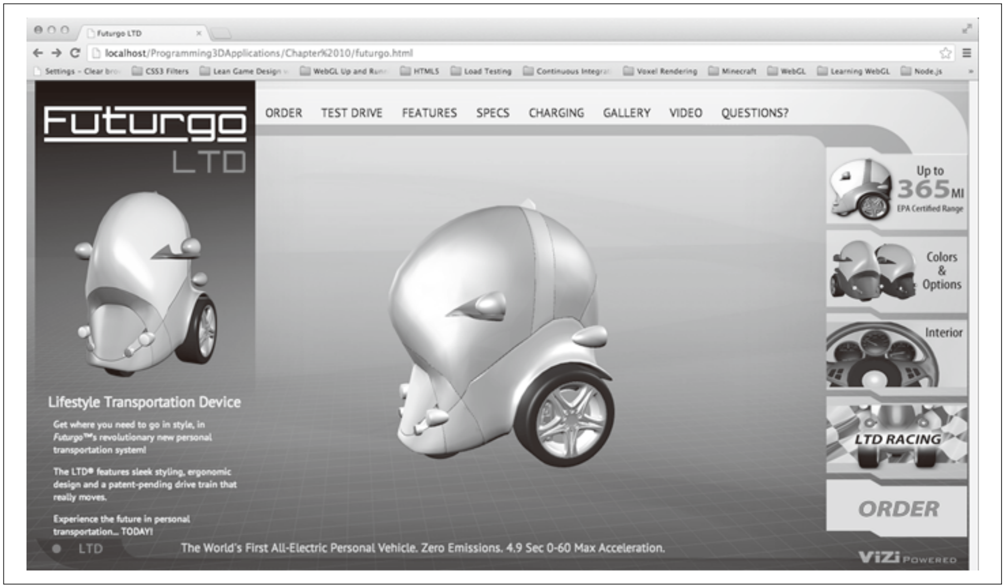


图 10-7：集成 Futurgo 模型到 HTML 页面中

注意页面中的所有元素是如何平滑整合的，这都拜浏览器合成引擎的强大能力所赐。每个页面元素是一个简单的 DIV 或几个嵌套的 DIV，有适当的顺序和 *z*-index 设置。3D 视图显示在页面其他元素的下层，所以界面看起来在上面。有些用户界面元素是透明的，这允许更多的 3D 场景显示出来。如果你看到的图片是彩色的，请注意 3D 场景背景所使用的紫色和灰色渐变，这直接源自 TC 的设计。我们简单地设置容器元素的 CSS `background` 属性，它就可以用于 WebGL canvas 的背景。这是非常强大的能力。最后，我们决定保留 `Vizi.Viewer` 所提供的灰色相框。（我承认这最开始是复制粘贴的意外，但我们喜欢这个效果所以决定保留它。）

应用的源码在 Chapter 10/futurgo.html、css/futurgo.css 和 Chapter 10/futurgo.js 中。相比之前的预览器，我们重构了一些代码。HTML 关注的只是标签，主要是 DIV，以及少量的脚本：页面加载代码、悬停后的处理程序、右边的界面标签。

页面加载代码如例 10-5 所示，它创建了一个新的 Futurgo 对象，向它传递了一个容器元素、一些加载完成及鼠标悬停和移出的回调函数。然后它调用 `Futurgo.go()`，这会加载 3D 场景，并启动运行循环。

例 10-5：Futurgo 页面加载代码

```
<script>

   var futurgo = null;
   var overlay = null;
   var overlayContents = null;
   var loadStatus = null;
   var part_materials = [];
```

```
$(document).ready(function() {

    initControls();
    overlay = document.getElementById("overlay");
    overlayContents = document.getElementById("overlayContents");
    loadStatus = document.getElementById("loadStatus");
    var container = document.getElementById("container");
    futurgo = new Futurgo({ container : container,
            loadCallback : onLoadComplete,
            loadProgressCallback : onLoadProgress,
            mouseOverCallback : onMouseOver,
            mouseOutCallback : onMouseOut,
         });

    loadStatus.style.display = 'block';
    futurgo.go();
}
);
```

Futurgo 对象处理了大部分麻烦的加载细节，页面代码中的加载回调只需要隐藏“Loading Scene...”的 DIV，具体代码请查看例 10-6。鼠标的回调函数 `onMouseOver()` 和 `onMouseOut()` 会在下一节介绍。

例 10-6：隐藏页面加载进程信息

```
    function onLoadComplete(loadTime)
    {
        // 隐藏加载状态栏
        loadStatus.style.display = 'none';
}
```

现在让我们来看看 `Futurgo` 类如何初始化查看器和加载场景，代码如例 10-7 所示（源码文件是 Chapter 10/futurgo.js）。

例 10-7：Futurgo 应用的查看器设置和文件加载代码

```
Futurgo = function(param) {

    this.container = param.container;
    this.loadCallback = param.loadCallback;
    this.loadProgressCallback = param.loadProgressCallback;
    this.mouseOverCallback = param.mouseOverCallback;
    this.mouseOutCallback = param.mouseOutCallback;
    this.part_materials = [];
    this.vehicleOpen = false;
    this.wheelsMoving = false;
}

Futurgo.prototype.go = function() {
    this.viewer = new Vizi.Viewer({ container : this.container,
        showGrid : true,
        allowPan: false, oneButton: true });
    this.loadURL(Futurgo.URL);
    this.viewer.run();
}
```

```
Futurgo.prototype.loadURL = function(url) {

    var that = this;

    var loader = new Vizi.Loader;
    loader.addEventListener("loaded", function(data) {
      that.onLoadComplete(data, loadStartTime); });
    loader.addEventListener("progress", function(progress) {
      that.onLoadProgress(progress); });

    var loadStartTime = Date.now();
    loader.loadScene(url);
}
```

到目前为止，这些代码大部分看起来应该都很熟悉。就像我们为预览工具所做的那样，我们创建了 `Vizi.Viewer` 和 `Vizi.Loader` 对象。当然，我们在创建查看器的时候设置了一些不同的参数（在代码中以粗体标注）。`allowPan` 控制了用户是否可以使用鼠标右键来上下左右移动物体。我们将它设置为 `false`，因为我们希望物体始终在场景的中间。`oneButton` 控制了是否鼠标右键也可以用来旋转模型。将它设置为 `true`，我们就可以使用鼠标左键或右键来进行旋转。

前面的代码将 Futurgo 模型加载到了页面中，看起来不错且可以进行交互了。接下来的一节我们会看看如何通过 3D 行为和交互来让它生动起来。

10.5 开发3D行为和交互

到目前为止我们创建的 Futurgo 应用已经非常有趣了。我们可以实时查看 3D 模型，还有良好的集成视觉效果，甚至可以使用鼠标控制模型。但它可以更完美，我们可以通过添加 3D 行为和交互来让它成为一个充分利用了 Web 实时图形能力的真正的交互应用。这包括使用透明渐变和一个旋转木马风格的旋转来自动动画模型加载，实现鼠标悬停时显示产品功能的更多信息，以及通过点击页面中的 2D 元素动态改变 3D 物体。

10.5.1 Vizi场景图API方法：findNode()和map()

本节介绍的行为需要遍历 `Vizi.Loader` 加载的 3D 内容场景图，所以我们可以给某些对象添加行为或鼠标交互。有时候我们需要通过名称或 id 来查找对象，有时候我们需要遍历场景图或其中一部分来查找某一类型的对象。Vizi 提供了一系列场景图 API 来做这些事情。可以向这些方法传递一个字符串标识符、一个 JavaScript 正则表达式，或者一个 JavaScript 对象类型（使用 `instanceof` 操作符来进行比较）。`findNode()` 和 `findNodes()` 方法返回所匹配的对象，`map()` 方法查找对象并在结果上调用一个函数。

- `findNode(query)`

  这个方法通过 query 查找一个节点（`Vizi.Object` 或 `Vizi.Component` 的实例）。query 可以是一个字符串标识符（例如“body2”）、一个对象类型（例如 `Vizi.Visual`），或者一个正则表达式（如 `/windows_front|windows_rear/`）。如果在 Vizi 场景图中有多个这样的节点，则返回第一个。

- `findNodes(query)`

  这个方通过 query 查找所有节点（`Vizi.Object` 或 `Vizi.Component` 的实例）。query 可以是一个字符串标识符（例如“body2”）、一个对象类型（例如 `Vizi.Visual`），或者一个正则表达式（如 `/windows_front|windows_rear/`）。

- `map(query, callback_function)`

  这个方法使用 `findNodes()` 来查找匹配搜索 query 的所有节点，然后在每个节点上调用回调函数。

> 你可以把 Vizi 场景图 API 方法当成类似 jQuery 查询的功能，尽管它们使用了完全不同的 query 方案。Vizi 使用的是字符串和 JavaScript 数据类型，而不是选择符。这是有意的设计，基于 Vizi 架构的对象及组件性质。

现在，让我们浏览一下 Futurgo 加载处理代码，来查看它如何添加行为，如例 10-8 中所示。

首先，`onLoadComplete()` 调用 `this.viewer.replaceScene(data)` 来将加载好的 Futurgo 场景添加到查看器中。在它的内部实现中，查看器不仅仅是将对象添加到它的场景图里，它还考虑了光源、相机及动画（前面介绍过），因此我们可以进行切换相机、播放动画等操作。然后，这个函数开始添加行为，甚至会启动其中的一些。这些行为的设置将在接下来的一节中逐一进行介绍。

例 10-8：在场景加载后添加行为到 Futurgo 应用中

```
Futurgo.prototype.onLoadComplete = function(data, loadStartTime)
{
    var scene = data.scene;
    this.viewer.replaceScene(data);

    // 为车窗添加进入时淡出的行为,并为鼠标经过行为添加选择器
    var that = this;
    scene.map(/windows_front|windows_rear/, function(o) {
        var fader = new Vizi.FadeBehavior({duration:2, opacity:.8});
        o.addComponent(fader);
        setTimeout(function() {
            fader.start();
        }, 2000);

        var picker = new Vizi.Picker;
        picker.addEventListener("mouseover", function(event) {
            that.onMouseOver("glass", event); });
        picker.addEventListener("mouseout", function(event) {
            that.onMouseOut("glass", event); });
        o.addComponent(picker);
    });

    // 自动旋转场景
    var main = scene.findNode("vizi_mobile");
    var carousel = new Vizi.RotateBehavior({autoStart:true,
      duration:20});
```

```
    main.addComponent(carousel);

    // 将所有的材质整合到一个集合中,以便改变颜色
    var frame_parts_exp =
  /rear_view_arm_L|rear_view_arm_R|rear_view_frame_L|rear_view_frame_R/;

    scene.map(frame_parts_exp, function(o) {
        o.map(Vizi.Visual, function(v) {
            that.part_materials.push(v.material);
        });
    });

    // 为鼠标经过行为添加选择器
    scene.map(/body2|rear_view_arm_L|rear_view_arm_R/, function(o) {
        var picker = new Vizi.Picker;
        picker.addEventListener("mouseover", function(event) {
            that.onMouseOver("body", event); });
        picker.addEventListener("mouseout", function(event) {
            that.onMouseOut("body", event); });
        o.addComponent(picker);
    });

    scene.map("wheels", function(o) {

        var picker = new Vizi.Picker;
        picker.addEventListener("mouseover", function(event) {
            that.onMouseOver("wheels", event); });
        picker.addEventListener("mouseout", function(event) {
            that.onMouseOut("wheels", event); });
        o.addComponent(picker);
    });

    // 通知页面加载完成
    if (this.loadCallback) {
        var loadTime = (Date.now() - loadStartTime) / 1000;
        this.loadCallback(loadTime);
    }
}
```

10.5.2 使用Vizi.FadeBehavior来动画透明度

我们希望将 Futurgo 模型的车窗变成半透明的，以便可以看见一些内饰的细节。我们可以在模型加载完后马上设置透明度，但有个渐变效果的话会更有趣。请看例 10-9。

例 10-9：给车窗添加渐变效果

```
var that = this;
scene.map(/windows_front|windows_rear/, function(o) {
    var fader = new Vizi.FadeBehavior({duration:2, opacity:.8});
    o.addComponent(fader);
    setTimeout(function() {
        fader.start();
    }, 2000);
```

Vizi.FadeBehavior 组件可以为任何可视对象及其内部的材质添加渐变效果。它的参数是 duration（单位是秒）和目标透明度值 opacity。在这个例子中，我们将在 2 秒的时间里渐变到 0.8 透明度（有点透明）。我们还在渐变前通过 setTimeout() 加了 2 秒的延迟。

为了了解 Vizi.FadeBehavior 做了什么，让我们看看它底层的实现。例 10-10 展示了 Vizi 源文件 src/behaviors/fadeBehavior.js 的部分代码。当行为启动的时候，它遍历所有包含的可视对象，查找当前的透明度。这会被用作 Tween.js 的初始值（请查阅第 5 章）。然后补间动画启动，运行一段时间。如果行为被激活，它的 evaluate() 方法会在每次运行循环的时候被调用。evaluate() 方法首先检查循环条件，如果需要的话会重新启动行为。然后，它遍历所有包含的可视对象，设置它们材质的透明度为新的补间值。我们通过一致的界面来为对象及它的组件轻松添加 FadeBehavior 这样的行为，并且适用于场景中所有的可视元素，这个功能很强大。

例 10-10：Vizi.FadeBehavior 的实现

```
Vizi.FadeBehavior.prototype.start = function()
{
    if (this.running)
        return;

    if (this._realized && this._object.visuals) {
        var visuals = this._object.visuals;
        var i, len = visuals.length;
        for (i = 0; i < len; i++) {
            this.savedOpacities.push(visuals[i].material.opacity);
            this.savedTransparencies.push(
              visuals[i].material.transparent);
            visuals[i].material.transparent = this.targetOpacity < 1 ?
              true : false;
        }
    }

    this.opacity = { opacity : this.savedOpacities[0] };
    this.opacityTarget = { opacity : this.targetopacity };
    this.tween = new TWEEN.Tween(this.opacity).to(this.opacityTarget,
      this.duration * 1000)
    .easing(TWEEN.Easing.Quadratic.InOut)
    .repeat(0)
    .start();

    Vizi.Behavior.prototype.start.call(this);
}

Vizi.FadeBehavior.prototype.evaluate = function(t)
{
    if (t >= this.duration)
    {
        this.stop();
        if (this.loop)
            this.start();
    }

    if (this._object.visuals)
```

```
    {
        var visuals = this._object.visuals;
        var i, len = visuals.length;
        for (i = 0; i < len; i++) {
            visuals[i].material.opacity = this.opacity.opacity;
        }
    }

}
```

10.5.3 使用Vizi.RotateBehavior来自动旋转内容

可以使用鼠标来旋转场景这点很好，但如果在用户不直接交互的时候就有点动画那就更好了。因此，我们在场景加载后设置自动旋转。Futurgo 的加载事件回调函数使用 `findNode()` 来找到 Futurgo 场景的根节点，然后给它添加 `RotateBehavior` 组件。旋转行为被设置为自动启动，20 秒循环一次。请看例 10-11。

例 10-11：添加一个 `Vizi.RotateBehavior` 来自动旋转内容

```
// 自动旋转场景
var main = scene.findNode("vizi_mobile");
var carousel = new Vizi.RotateBehavior({autoStart:true,
  duration:20});
main.addComponent(carousel);
```

10.5.4 使用Vizi.Picker来实现鼠标悬停时显示信息

鼠标悬停是一个提供页面中元素详细信息的好方法。我们可以使用第 9 章介绍过的 `Vizi.Picker` 组件，来拓展实现单个对象在 3D 场景中鼠标悬停时显示信息的想法。这个组件提供了通用的鼠标处理方法，当鼠标悬停在特定对象上的时候触发事件。

让我们回到给车窗添加淡出行为的代码。注意这段代码还添加了 picker 组件，请看例 10-12 中粗体标注的代码行。通过相同的方式，我们给车身各个部分及轮子添加了 picker。每个回调函数使用了不同的标签——“glass”“body”及“wheels”，它被传递给应用以判断出悬停在哪个部分。

例 10-12：添加 `Vizi.Picker` 组件来实现悬停

```
// 为车窗添加进入时淡出的行为,并为鼠标经过行为添加选择器
var that = this;
scene.map(/windows_front|windows_rear/, function(o) {
    var fader = new Vizi.FadeBehavior({duration:2, opacity:.8});
    o.addComponent(fader);
    setTimeout(function() {
        fader.start();
    }, 2000);

    var picker = new Vizi.Picker;
    picker.addEventListener("mouseover", function(event) {
        that.onMouseOver("glass", event); });
    picker.addEventListener("mouseout", function(event) {
        that.onMouseOut("glass", event); });
```

```
        o.addComponent(picker);
    });
...
    // 为鼠标经过行为添加选择器
    scene.map(/body2|rear_view_arm_L|rear_view_arm_R/, function(o) {
        var picker = new Vizi.Picker;
        picker.addEventListener("mouseover", function(event) {
            that.onMouseOver("body", event); });
        picker.addEventListener("mouseout", function(event) {
            that.onMouseOut("body", event); });
        o.addComponent(picker);
    });

    scene.map("wheels", function(o) {

        var picker = new Vizi.Picker;
        picker.addEventListener("mouseover", function(event) {
            that.onMouseOver("wheels", event); });
        picker.addEventListener("mouseout", function(event) {
            that.onMouseOut("wheels", event); });
        o.addComponent(picker);
    });
```

辅助方法 `Futurgo.onMouseOver()` 和 `Futurgo.onMouseOut()` 只是调用在 `Futurgo` 类初始化时传入的 `onMouseOver` 和 `onMouseOut` 回调函数（参见例 10-5）。

悬停行为如图 10-8 所示。当鼠标在对象上悬停时，会在场景窗口的右边与鼠标指针 *y* 轴近似的位置显示一个 DIV。

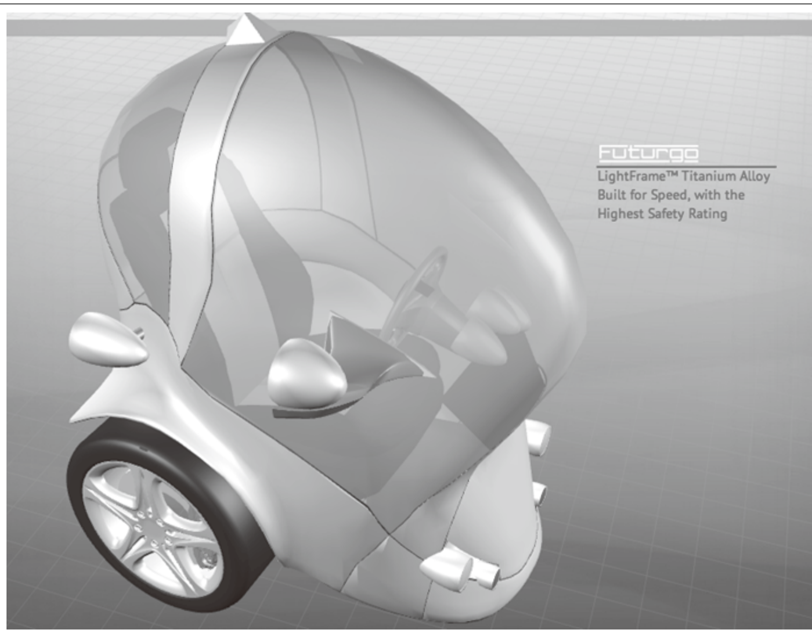


图 10-8：悬停提供了产品的更多信息

10.5.5 使用用户界面来控制动画

我们还可以使用 HTML 页面中的 2D 界面元素来控制 3D 场景内的行为。当我们点击右边的 Interior 或 LTD Racing 标签后，它会启动 Futurgo 中的动画。例 10-13 中展示了在 HTML 页面内设置点击回调，然后调用 Futurgo 对象中方法的代码。这些方法会调用查看器的 `playAnimation()` 和 `stopAnimation()` 方法来完成工作。不过需要注意一点：我们需要在第一次点击 Interior 的时候打开车窗，再次点击的时候关闭。我们并不是单独创建了打开和关闭的动画，而是简单地在第二次点击的时候反着播放动画。请看 Futurgo 的 `playCloseAnimations()` 方法，它向查看器多传入了两参数。第二个参数 `loop` 设置为 `false`，第三个参数 `reverse` 设置为 `true`。和 Tween.js 一样，Vizi 的动画引擎内置了从不同方向播放动画的能力。

例 10-13：通过用户界面来控制动画

```
Futurgo.prototype.playOpenAnimations = function() {
    this.playAnimation("animation_window_rear_open");
    this.playAnimation("animation_window_front_open");
}

Futurgo.prototype.playCloseAnimations = function() {
    this.playAnimation("animation_window_rear_open", false, true);
    this.playAnimation("animation_window_front_open", false, true);
}

Futurgo.prototype.toggleInterior = function() {
    this.vehicleOpen = !this.vehicleOpen;
    var that = this;
    if (this.vehicleOpen) {
        this.playOpenAnimations();
    }
    else {
        this.playCloseAnimations();
    }
}

Futurgo.prototype.playWheelAnimations = function() {
    this.playAnimation("animation_wheel_L", true);
    this.playAnimation("animation_wheel_R", true);
    this.playAnimation("animation_wheel_front", true);
}

Futurgo.prototype.stopWheelAnimations = function() {
    this.stopAnimation("animation_wheel_L");
    this.stopAnimation("animation_wheel_R");
    this.stopAnimation("animation_wheel_front");
}

Futurgo.prototype.toggleWheelAnimations = function() {
    this.wheelsMoving = !this.wheelsMoving;
    if (this.wheelsMoving) {
        this.playWheelAnimations();
    }
    else {
```

```
        this.stopWheelAnimations();
    }
}
```

10.5.6　使用颜色选择器来改变颜色

任何遵循业内惯例的产品展示页面都必须提供改变颜色的能力，所以 Futurgo 也不例外。我们整合了一个 jQuery 颜色选取控件，供用户为他们的爱车从 1600 万种颜色中选择一种。改变颜色选取控件中的颜色会立刻更换 Futurgo 车身的颜色，如图 10-9 所示。

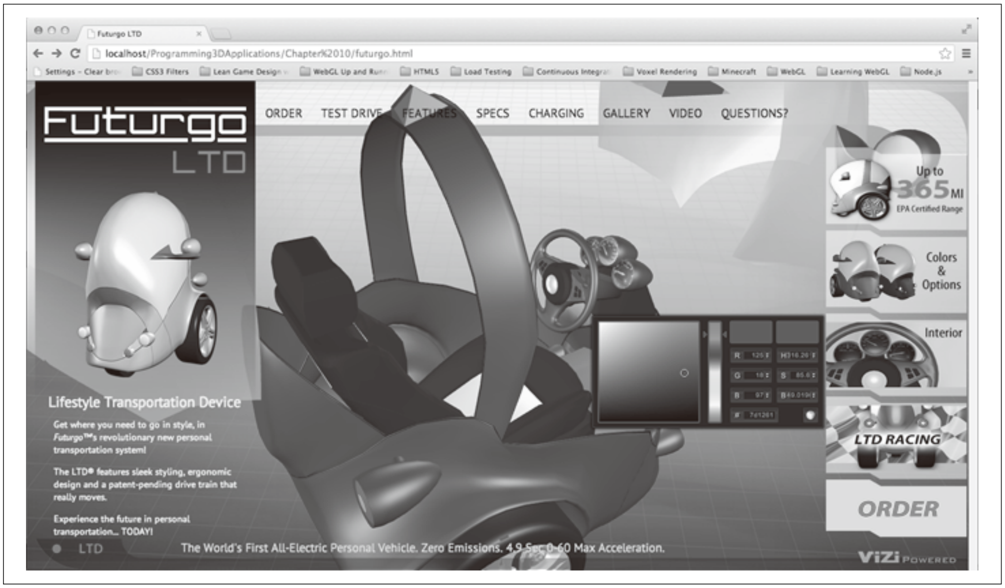


图 10-9：使用颜色选取器改变颜色

让我们查看之前连接鼠标事件与对象的代码，它在 Futurgo 的 `onLoadComplete()` 方法里。在例 10-14 中，注意 `map()` 的递归调用：对于通过正则表达式查找到的每一个节点，我们将查找它的所有可视部分，并将它的 Three.js 材质插入到数组 `part_materials` 里。

例 10-14：存储 Futurgo 车身材质的代码

```
        var frame_parts_exp =
    /rear_view_arm_L|rear_view_arm_R|rear_view_frame_L|rear_view_frame_R/;

      scene.map(frame_parts_exp, function(o) {
          o.map(Vizi.Visual, function(v) {
              that.part_materials.push(v.material);
          });
      });
```

现在我们保存了这些材质，我们可以通过用户界面来控制它们。`Futurgo` 定义了另外两个方法，分别用于获得和设置车身的颜色，它被 HTML 页面中的颜色选择器所使用。请看例 10-15。

例 10-15：获得和设置 Futurgo 车身颜色的代码

```
Futurgo.prototype.getBodyColor = function() {
    var color = '#ffffff';
    if (this.part_materials.length) {
        var material = this.part_materials[0];
        if (material instanceof THREE.MeshFaceMaterial) {
            color = '#' + material.materials[0].color.getHexString();
        }
        else {
            color = '#' + material.color.getHexString();
        }
    }

    return color;
}

Futurgo.prototype.setBodyColor = function(r, g, b) {

    // 将hex rgb颜色值转换为浮点数
    r /= 255;
    g /= 255;
    b /= 255;

    var i, len = this.part_materials.length;
    for (i = 0; i < len; i++) {
        var material = this.part_materials[i];
        if (material instanceof THREE.MeshFaceMaterial) {
            var j, mlen = material.materials.length;
            for (j = 0; j < mlen; j++) {
                material.materials[j].color.setRGB(r, g, b);
            }
        }
        else {
            material.color.setRGB(r, g, b);
        }
    }
}
```

`getBodyColor()` 返回了车身材质当前的漫反射颜色。尽管在列表中有好几种材质，我们实际上只需要第一个的值，因为（理论上）它们是一样的。我们返回 CSS 样式的十六进制字符串。颜色选择器使用这个值在弹出层出现前初始化色卡和输入值。

而对于 `setBodyColor()`，我们必须遍历数组中所有的材质，设置它们的漫反射颜色。回忆一下，在 Three.js 中，有些对象有一个 `THREE.MeshFaceMaterial` 类型的材质，它实际上是一个对象的每个面的材质的数组。在上面代码中考虑到了这一点。`setBodyColor()` 方法的 RBG 值是从颜色选择器中传入的十六进制颜色（即从 `0` 到 `255` 的整数），而 Three.js 需要从 `0` 到 `1` 的浮点数，所以这个方法进行了转换。

10.6　小结

本章描述了构建一个简单但完全可用的 3D Web 应用的详细步骤。我选择了一个 3D 产品页面作为例子，因为它能充分说明关键概念。在简要介绍了视觉设计流程后，我们学习了如何在专业 DDC 工具 Maya 中创建内容，然后转换为适合 Web 的 glTF 格式。之后，我们使用 Vizi 框架来开发预览和测试 3D 内容的工具。再然后我们介绍了如何将内容集成到应用页面中。最后，我们添加了几个行为和交互，进一步改进，让它更有趣和更易用。

开发 3D Web 应用的过程很复杂，不过有了合适的工具和知识，你会发现它是很好实现的。下一章会介绍一些新的 3D 行为和交互，但你在本章学到的全部流程和技术都可以应用到后续所有的开发中。

第 11 章

开发一个3D环境

第 10 章介绍的技术覆盖了许多使用场景。3D 模型可以用来构建营销、咨询、娱乐等交互式内容。但许多 3D 应用有更多需求。如果想开发一个身临其境的游戏、一个建筑查看器，或者一个交互式训练系统，我们需要学习如何创建 3D 环境，而其中有多个物体以及更复杂的交互类型。

在本章中，我们会开发一个 3D 环境，其中有真实的风景、移动的物体，还能让用户通过交互式地控制相机来在场景中导航。我们将扩展在第 10 章中开发的内容，创建一个虚拟的城市，在其中试驾 Futurgo 概念车。图 11-1 展示了这个应用。

Futurgo 停在城市街道上，已经准备好供人试驾了。场景横跨了几个街区，远处若隐若现的摩天大楼连接着灰蒙蒙的天空，近处的办公大楼上反射出了它们的影像。你可以使用鼠标点击和拖拽来上下左右查看，同时使用键盘的方向键来上下左右移动。走近 Futurgo，点击它来坐进去兜风。这个世界或许看起来有些空旷，但我们坐在自己的交通工具中很安全。

在你的浏览器中打开文件 Chapter 11/futurgoCity.html 来试试。在构建这个应用的过程中，我们会讨论以下话题。

- *创建环境素材*

  使用道路、建筑和公园来组装真实的 3D 城市场景。

- *预览和测试*

  向上一章开发的预览工具中添加功能。对于这个项目而言，我们需要一个预览工具，能加载多个文件到一个场景中，能够显示场景图的结构，并允许我们查看不同物体的属性。这一切都是为开发应用所做的准备工作。

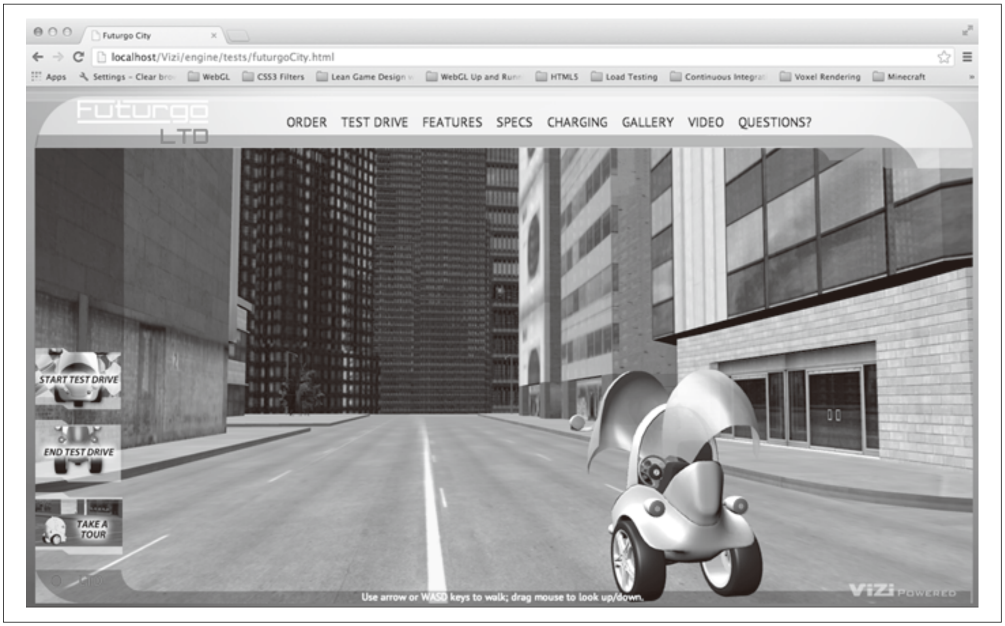


图 11-1：在一个 3D 环境中的 Futurgo 概念车

- 创建一个 3D 背景

  使用 skybox 创建一个真实的天空背景，这是一个带纹理的立方体，放在背景的无限远处。同样的 skybox 纹理还用在城市建筑和车辆上，作为反射天空背景的立方体环境贴图。

- 集成 3D 到应用中

  管理加载多个模型到同一个应用的细节，调整车模型的灯光、位置和其他属性来匹配周围环境。

- 实现第一人称导航

  为用户提供通过鼠标和键盘在场景中环顾四周及到处移动的方法，并实现碰撞机制使得用户不会穿过实体物体。

- 使用多个相机

  切换相机，允许用户从不同角度查看场景，使用不同方式进行浏览。

- 创建定时的动画过渡

  使用计时器和动画技术来在用户进入和离开车的时候，创建动画序列。

- 对象行为脚本

  使用 Vizi 框架创建自定义组件，来控制 Futurgo 车的行为和外观。
  使用声音通过添加 HTML5 声音元素来增强环境。

- 渲染动态纹理

  通过使用 2D Canvas API 来以编程的方式更新 3D 物体的纹理，为用户提供实时反馈。

我们在本章中创建的虚拟场景很简单。一个典型的游戏或其他 3D 环境会有更多的物体及更复杂的交互。不过这里介绍的技术，为学习如何开发更复杂的场景提供了良好的开端。

11.1 创建环境素材

为了开发 3D 环境，我再次和艺术家 TC Chang 合作。创建城市背景素材是很费时的，所以 TC 和我决定寻找现成的模型。我们在 TurboSquid 上找到了一个优秀的候选作品，如图 11-2 所示。

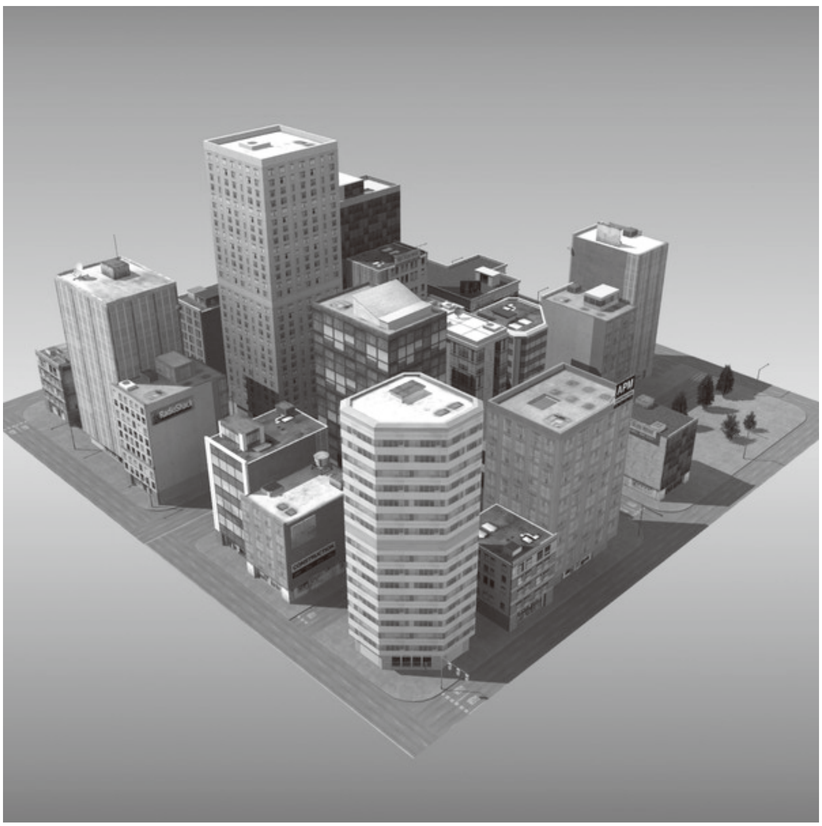


图 11-2：ES3DStudios 创建的城市模型；图片由 TurboSquid 提供

这个城市模型是使用 Lightwave 建模工具（https://www.lightwave3d.com/overview/）创建的。作者已经将它转换成了多种格式，包括 Autodesk Maya。我们购买和下载模型后，TC 将它导入 Maya 中，为在应用中使用做准备。这个模型有充满细节和纹理的建筑，但没有光源。TC 添加了三个光源，这样模型就可以使用了。将它导出为 COLLADA 格式并加载到预览工具（参见下一节）中后，我们发现了一个有关纹理贴图透明度的小问题。不过总体来说，在应用中使用这个模型只需做非常少的额外工作。

11.2 预览和测试环境

为了测试城市场景这样的复杂模型，我们需要一个类似第 10 章创建的预览工具，但它需要更多功能。新的增强版预览工具如图 11-3 所示，它增加了以下几个功能。

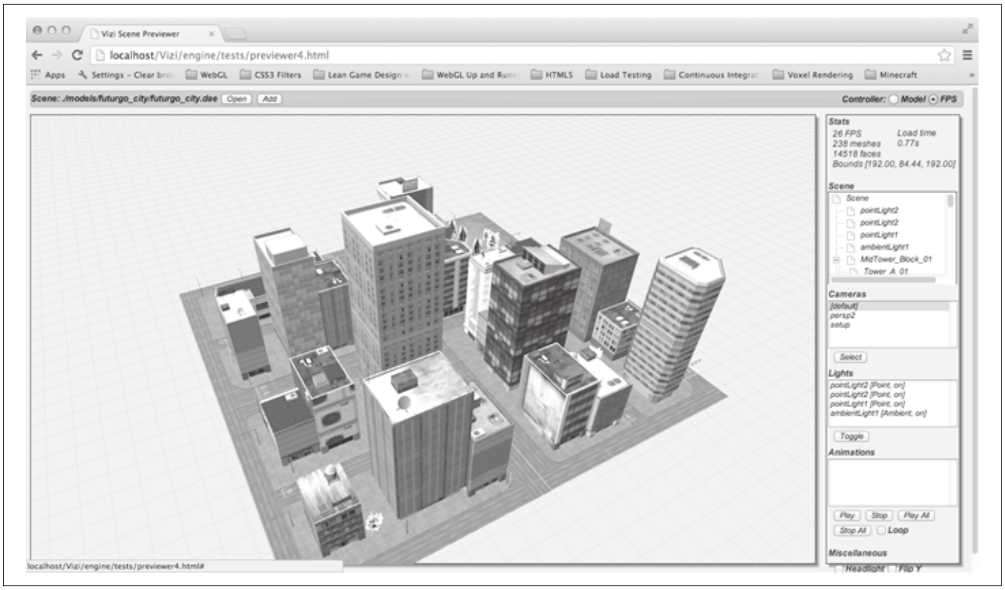


图 11-3：显示在 Vizi 预览工具中的城市环境

- 多种浏览模式

  能够使用相机朝着一个单一模型的中心来浏览，或者用相机朝着地面来浏览场景。

- 场景图检查

  一个场景图的树形结构视图，用来显示对象的名称和父子关系。

- 对象检查

  一个弹出层属性列表显示每个对象的详情，包括变换信息、网格统计信息、材质属性，以及相机和光源的参数。

- 边界框显示

  一个线框盒显示在所选对象的周围，还有一个选项可以显示场景中所有物体的边界框。

- 多物体预览

  能够在同一个预览中加载多个物体，用来观察和测试它们合成后的效果。

打开文件 Chapter 11/previewer.html。点击 Open 按钮会出现一个文件对话框，打开文件 ../models/futurgo_city/futurgo_city.dae。使用相机列表，选择标记为 [default] 的相机来自由导航。使用鼠标来旋转，使用触摸板或滚轮来缩放，你可以检查城市模型。（注意其他相机不允许你使用鼠标自由导航，只有 [default] 可以。）

11.2.1 以第一人称模式预览场景

当你旋转和缩放城市模型的时候，或许会注意到相机不能完全到达场景的地面（街道）。这是因为预览工具默认设计为将模型当成一个单一对象，相机指向的是对象的几何中心。单一对象的方案并不适合环境，所以我们添加了另一种浏览模式作为辅助。

在预览工具界面的右上角，有一个单选按钮组来让你切换浏览模式。这个组的标签为控制器（Controller），有两种不同的相机控制器模式：模型模式（Model）和第一人称模式（FPS）。我们的城市应用将使用第一人称控制器，它是设计用来在环境中导航的，而不是查看一个单独的模型。（第一人称导航将在本章后面详细地介绍。）点击 FPS 按钮，相机会下降，旋转的中心现在在街道上了，你可以缩放到街道表面上。

注意，预览工具的 FPS 模式并不是真正使用第一人称导航模式来浏览模型。它只是简单地将相机放到你在真实第一人称模式中相似的位置上，它更适合预览整个环境。预览工具内部依然使用一个模型控制器，因此我们可以快速缩放和旋转整个模型。换句话说，有时候我们希望将场景当成一个单一模型来方便操作，有时候我们希望模拟在应用中浏览环境的时候会看到的效果。预览工具中的 FPS 按钮这一简单的小技巧能做到两全其美。

11.2.2 检查场景图

因为我们的场景变得更加复杂，所以我们需要预览工具能更精细地查看它们。以城市场景为例，它由超过 200 个单独的网格组成，显示在预览工具的场景状态栏中。为了在应用中程序化交互，我们需要找到独立对象的名称、大小、位置、类型（如网格、相机或光源）等属性，以及对象的层级结构。这对于从第三方获得的模型尤其重要，因为在这种情况下，我们不能和创建内容的艺术家进行密切沟通。

一个粗暴的方式是在文本编辑器中打开 COLLADA 或 glTF 文件来检查场景图，查找指示类型的具体文本。但这对绝大多数开发者而言都是令人发狂的体验，它需要技术细节知识，知晓这些文件格式是如何组织的。（我个人对这两种文件格式都非常了解，但我没耐心浏览大量的文本，在其中大海捞针般地寻找。）一个更好的方式是由预览工具来为我们展现这个信息。

增强版的预览工具包含一个新的面板 Scene，并且有一个列表框来展示场景图层级结构的树状图。你最好花些时间滚动列表，点击加号和减号来展开和收起层级，看看它是怎么组织的：在顶层有几个光源，接下来是一个名为 `MidTower_Block_01` 的组，后面是几个相机。注意组旁边的加号，如果你点击它，这个组会展开显示子层级，有名称为 `Tower_A_01`、`Roof_Detail_01` 等的对象。这些组中有些会自展开以显示子对象。

有了在场景中查看节点名称和层级关系的能力，我们就可以确定在运行应用时，将要向哪个对象添加交互以及其他一些细节。比如，在应用中加载完场景后，我们将添加环境贴图来反射天空背景，但只用于建筑而不是道路或公园上。浏览场景层级结构显示，建筑的名称都以“Tower”或“Office”开头，所以我们就可以使用 `Vizi.Object.map()` 这个场景图 API 方法，来查找所有匹配这个正则表达式的对象，并修改它们的材质。我们将在本章稍后介绍实现的代码。

预览工具中使用的树状图是用一个叫 dynatree（http://code.google.com/p/dynatree/）的 jQuery 插件实现的。例 11-1 展示了使用多种参数初始化树状图控件的代码，和设置当项被单击或双击时的回调函数的代码。预览工具的源码可以在文件 Chapter 11/previewer.html 中找到。

例 11-1：初始化 dynatree 树状图控件

```
function initSceneTree(viewer) {
    // 初始化<div>元素中的树
    $("#scene_tree").dynatree({
        imagePath: "./images/previewer_skin/",
        title: "Scene Graph",
        minExpandLevel: 2,
        selectMode: 1,
        onDblClick: function(node) {
            openSceneNode(viewer, node);
        },
        onActivate: function(node) {
            selectSceneNode(viewer, node);
            if (infoPopupVisible) {
                openSceneNode(viewer, node);
            }
        },
        onDeactivate: function(node) {
        },
        onFocus: function(node) {
        },
        onBlur: function(node) {
        },
    });
}
```

现在我们来分析如何在场景文件加载好后，基于场景图的内容填充树状图控件。首先，我们在加载回调中添加了一行代码，来调用辅助函数 `updateSceneTree()`。

```
function onLoadComplete(data, loadStartTime)
  {
      // 隐藏加载状态栏
      var loadStatus = document.getElementById("loadStatus");
      loadStatus.style.display = 'none';

      viewer.replaceScene(data);

      var loadTime = (Date.now() - loadStartTime) / 1000;
      var loadTimeStats = document.getElementById("load_time_stats");
      loadTimeStats.innerHTML = "Load time<br>" + loadTime.toFixed(2) + "s"
      // Vizi.System.log("Loaded " + loadTime.toFixed(2) + " seconds.");

      updateSceneTree(viewer);
      updateCamerasList(viewer);
      updateLightsList(viewer);
      updateAnimationsList(viewer);
      updateMiscControls(viewer);

      if (viewer.cameraNames.length > 1) {
```

```
            selectCamera(1);
        }

        addRollovers(viewer, data.scene);
    }
```

updateSceneTree() 做了几件事情。首先，它在树状图的根节点调用 removeChildren() 来重新初始化树状图控件，因为它在之前浏览其他场景的时候可能已经被填充过。然后，它调用另一个函数 buildSceneTree() 来遍历场景图，并填充树状图控件的内容。注意这个调用封装到了一个 setTimeout() 中，稍微延迟了一下。这个延迟是为了更好的用户体验。使用 dynatree 构建树状图需要一些时间，而我们不想拖慢场景的初始渲染。所以我们在开始的时候放了一个占位信息，当 timeout 触发的时候删掉。

```
    function updateSceneTree(viewer) {

        // 示例:用代码添加一个层级hierarchic branch
        // 这里展示了我们如何用程序添加树节点
        var rootNode = $("#scene_tree").dynatree("getRoot");
        rootNode.removeChildren();
        var initMessage = rootNode.addChild({
            title: "Initializing...",
            isFolder: false,
        });

        setTimeout(function() {
            rootNode.removeChild(initMessage);
            rootNode.expand(false);
            var i, len = viewer.scenes.length;
            for (i = 0; i < len; i++) {
                buildSceneTree(viewer.scenes[i], rootNode);
            }
        }, 1000);
    }
```

填充场景树状图的代码实际上很简单，buildSceneTree() 函数的源码在文件 Chapter 11/sceneTree.js 中。例 11-2 完整展示了它。

例 11-2：填充场景树状图

```
sceneTreeMap = {};

buildSceneTree = function(scene, tree) {

    function build(object, node, level) {

        var noname = level ? "[object]" : "Scene";

        var childNode = node.addChild({
            title: object.name ? object.name : noname,
            expand: level <= 1,
            activeVisible:true,
            vizi:object,
        });
```

```
        sceneTreeMap[object._id] = childNode;

        var i, len = object._children.length;
        for (i = 0; i < len; i++) {
            build(object._children[i], childNode, level+1);
        }
    }

    build(scene, tree, 0);

}
```

首先，我们初始化一个全局对象 sceneTreeMap，它将 Vizi 场景图中的 Vizi 对象与树状图控件中的项关联起来。我们将用它来支持点击场景中的某个对象时，高亮控件中关联的项。

在 buildSceneTree() 函数内部，我们定义了一个内嵌函数 build()，它会递归添加项到树控件中。对于 Vizi 场景图中的每个对象，build() 函数会调用 node.addChild() 创建一个新的树控件节点。node.addChild() 方法基于所提供的参数创建了一个新的项。

title 是项显示的标签，expend 标识指示初始显示时是否展开项。我们只在场景图顶层展开。activeVisible 定义了如果激活某一项（即从代码中选择它，比如当在场景中点击关联的对象时），树状图控件就滚动到并选定它。传递的最后一个参数是 vizi，它是 Vizi 场景图对象，将会在用户点击树状图控件中的一项的时候用到。当点击的时候，预览工具会使用黄色线框盒高亮它，当双击的时候会显示一个相关节点属性的弹出层（将在下一节介绍）。

当树控件的项创建好后，我们将它添加到 sceneTreeMap 对象中供后续使用，如果它有子节点，就会递归调用 build() 来添加树控件项。

> 创建一个好的基于 HTML 的树状图需要很多工作。幸运的是，dynatree 的开发者让我们免去了这个痛苦。dynatree 允许你立刻创建一个任意层级的 HTML 树状图，它还有一个功能强大、易于使用的 API 来创建 / 修改 / 删除项，并且它支持完全自定义样式。dynatree 的代码托管在 Google code（http://code.google.com/p/dynatree/）上。

11.2.3 检查对象属性

预览工具允许我们检查每个对象的属性。双击场景树状图的一个对象，就会弹出一个带标签的 jQuery 对话框（或者*属性页单*）来显示详情。如图 11-4 所示，其中的属性页单显示了名为 Tower_D_01 的对象的属性。它包含三个标签：一个显示变换信息（位置、旋转和缩放）；一个显示几何体的详情，包括网格中顶点和面的数量，以及它的边界框；一个显示材质信息，包括着色模型、颜色、纹理贴图的图片名称。

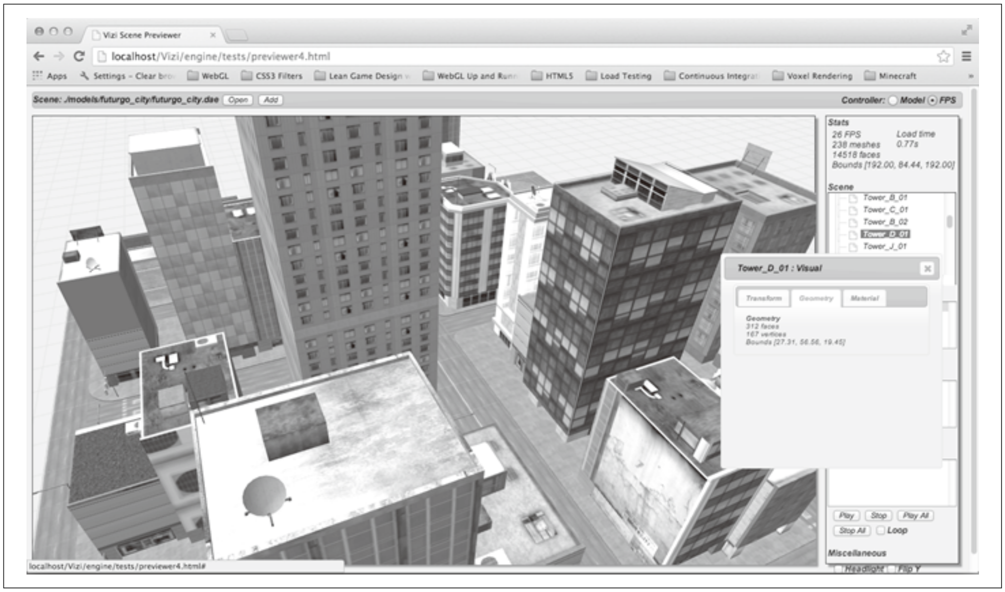


图 11-4：使用预览工具来检查对象属性

预览工具还允许我们在 3D 场景中点击对象的时候查看其属性。如果你在属性页单已经打开的时候点击对象，它的内容会替换为新对象的属性。如果属性页单没有打开，你可以双击对象来弹出这个对话框以显示对象的属性。

例 11-3 展示了在 3D 场景中添加点击监听的代码（源文件为 Chapter 11/previewer.html），这使得用户可以选择单个物体。函数 `addRollovers()` 使用 Vizi 场景图 API 的 `map()` 方法来查找场景中的每一个对象，并创建一个新的 `Vizi.Picker` 对象来添加鼠标事件响应。代码添加了对鼠标 down（按下）、up（松开）、over（悬停）、双击（double-click）事件的响应。

例 11-3：实现在场景中选择对象

```
function addRollover(viewer, o) {
    var picker = new Vizi.Picker;
    picker.addEventListener("mouseover", function(event) {
        onPickerMouseOver(viewer, o, event); });
    picker.addEventListener("mouseout", function(event) {
        onPickerMouseOut(viewer, o, event); });
    picker.addEventListener("mouseup", function(event) {
        onPickerMouseUp(viewer, o, event); });
    picker.addEventListener("dblclick", function(event) {
        onPickerMouseDoubleClick(viewer, o, event); });
    o.addComponent(picker);
}

function addRollovers(viewer, scene) {
    scene.map(Vizi.Object, function(o) {
        addRollover(viewer, o);
    });
}
```

鼠标松开事件的监听代码，是用来检测单击的，双击的检测在后面。它们几乎是一样的。首先，我们检查事件的按键代号，因为预览工具只支持使用鼠标左键选择。如果是左键，我们调用 `Vizi.Viewer` 的 `highlightObject()` 方法。它会在被点击对象周围画一个黄色的线框盒（我们会在下一节介绍边界框高亮的实现细节）。

接着，我们高亮树状图中关联的项，使用 Vizi 对象的 `_id` 属性（这个属性会由 Vizi 引擎在对象创建的时候自动生成）来作为索引查找树中相关的项，并高亮它。最后，如果是单击且属性页单已经可见了（通过 `infoPopupVisible` 变量来标识），我们调用 `openSceneNode()` 方法，它是一个使用新选择节点来重新填充属性页单的辅助方法。对于双击的情况，我们直接调用 `openSceneNode()`，如果对话框没显示的就会将它弹出来。

```
function onPickerMouseUp(viewer, o, event) {
    if (event.button == 0) {
        viewer.highlightObject(o);
        node = selectSceneNodeFromId(viewer, o._id);
        if (node && infoPopupVisible) {
            openSceneNode(viewer, node);
        }
    }
}

function onPickerMouseDoubleClick(viewer, o, event) {
    if (event.button == 0) {
        viewer.highlightObject(o);
        node = selectSceneNodeFromId(viewer, o._id);
        openSceneNode(viewer, node);
    }
}
```

11.2.4 显示边界框

预览工具使用边界框来实现两个功能：高亮所选的对象；如果你选择了界面中的“显示所有对象边界框”选项，就显示所有对象的边界框。

为了高亮所选对象，`Vizi.Viewer` 提供了一个叫 `highlightObject()` 的方法。例 11-4 展示了它的实现。首先，查看器移除当前对象的高亮（如果有的话），然后计算新对象的边界框，使用它来创建一个黄色的线框盒围绕对象。

这里有几个需要注意的地方，请看粗体标出的代码。我们创建了一个 `Vizi.Decoration` 对象来包含边界框立方体。`Vizi.Decoration` 是一个特殊的 `Vizi.Visual` 子类，框架使用它来渲染你可以看见但不进行交互的内容。它不会干扰选取或碰撞。之后，我们添加 decoration 到对象的*父节点*上，而不是这个对象中。任意对象的边界框都是在父节点的坐标系统中计算的，所以我们需要将它作为父节点的子节点添加到场景图中，以便正确地进行变换。

例 11-4：创建所选对象的高亮框

```
Vizi.Viewer.prototype.highlightObject = function(object) {

   if (this.highlightedObject) {
```

```
        this.highlightedObject._parent.removeComponent(
            this.highlightDecoration);
    }

    if (object) {
        var bbox = Vizi.SceneUtils.computeBoundingBox(object);

        var geo = new THREE.CubeGeometry(bbox.max.x - bbox.min.x,
                bbox.max.y - bbox.min.y,
                bbox.max.z - bbox.min.z);

        var mat = new THREE.MeshBasicMaterial({color:0xaaaa00,
            transparent:false,
            wireframe:true, opacity:1})

        var mesh = new THREE.Mesh(geo, mat);
        this.highlightDecoration = new Vizi.Decoration({object:mesh});
        object._parent.addComponent(this.highlightDecoration);

        var center = bbox.max.clone().add(bbox.min)
              .multiplyScalar(0.5);
        this.highlightDecoration.position.add(center);
    }

    this.highlightedObject = object;
}
```

预览工具允许你查看所有对象的边界框。在右下角的 Miscellaneous 栏中，有一个名为 Boxes 的复选框。点击它，显示所有对象的边界框，你可以看到类似图 11-5 所示的绿色线框。

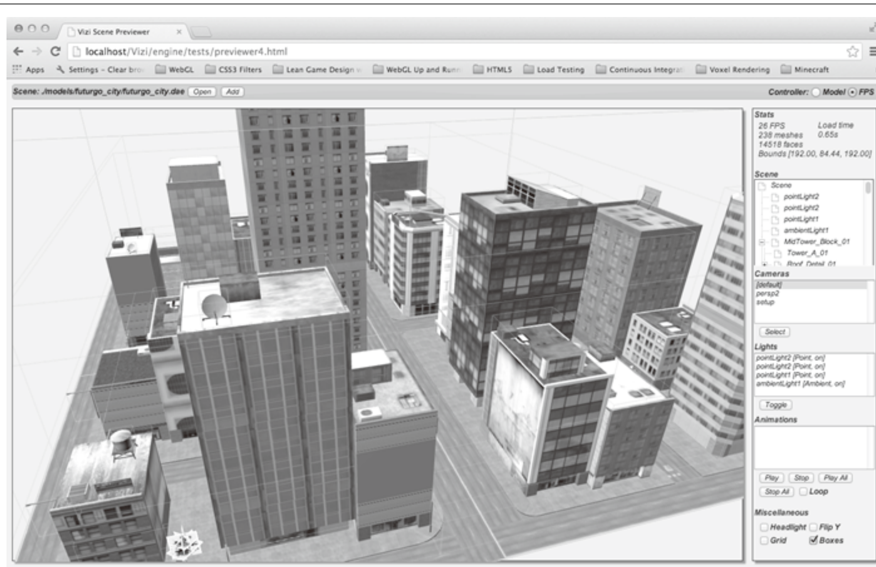


图 11-5：预览工具显示场景中所有对象的边界框（另见彩插图 11-5）

显示每个对象边界框的代码类似创建高亮框的代码，只是这次我们使用 Vizi 场景图 API 的 `map()` 方法来将其应用到场景图中的所有对象上。请看例 11-5。

例 11-5：为场景中的所有对象创建边界框

```
        this.sceneRoot.map(Vizi.Object, function(o) {
            if (o._parent) {
                var bbox = Vizi.SceneUtils.computeBoundingBox(o);

                var geo = new THREE.CubeGeometry(bbox.max.x - bbox.min.x,
                        bbox.max.y - bbox.min.y,
                        bbox.max.z - bbox.min.z);
                var mat = new THREE.MeshBasicMaterial(
                     {color:0x00ff00, transparent:true,
                          wireframe:true, opacity:.2})
                var mesh = new THREE.Mesh(geo, mat)
                var decoration = new Vizi.Decoration({object:mesh});
                o._parent.addComponent(decoration);

                var center = bbox.max.clone().add(bbox.min)
                     .multiplyScalar(0.5);
                decoration.position.add(center);
                decoration.object.visible = this.showBoundingBoxes;
            }
        });
```

现在，当用户点击 Boxes 复选框来开关这个功能的时候，预览工具调用查看器的 `setBoundingBoxesOn()` 方法。这个方法使用 `map()` 来查找每个类型为 `Vizi.Decoration` 的对象，通过设置它的 `visible` 属性来开关它的可见性。

```
    Vizi.Viewer.prototype.setBoundingBoxesOn = function(on)
    {
        this.showBoundingBoxes = !this.showBoundingBoxes;
        var that = this;
        this.sceneRoot.map(Vizi.Decoration, function(o) {
            if (!that.highlightedObject || (o != that.highlightDecoration)) {
                o.visible = that.showBoundingBoxes;
            }
        });
    }
```

11.2.5　预览多个对象

当你使用多个对象来构建一个环境时，能够将它们放到一起预览和测试是至关重要的。我们需要保证对象的比例是一致的，相对位置正确，光照合适等，尤其是对象来自多个地方，由不同的艺术家创建，托管在不同的模型共享网站上时。

让我们将 Futurgo 车模型放到城市场景中测试这些属性。点击顶部菜单栏的 Add 按钮，在文件选择框中选择 ../models/futurgo_mobile/futurgo_mobile.json。（请保证你使用的是默认相机，并且正对着场景中间的主路，这是模型将出现的地方。）模型会在场景的中间出现。放大来近距离观察，如图 11-6 所示。

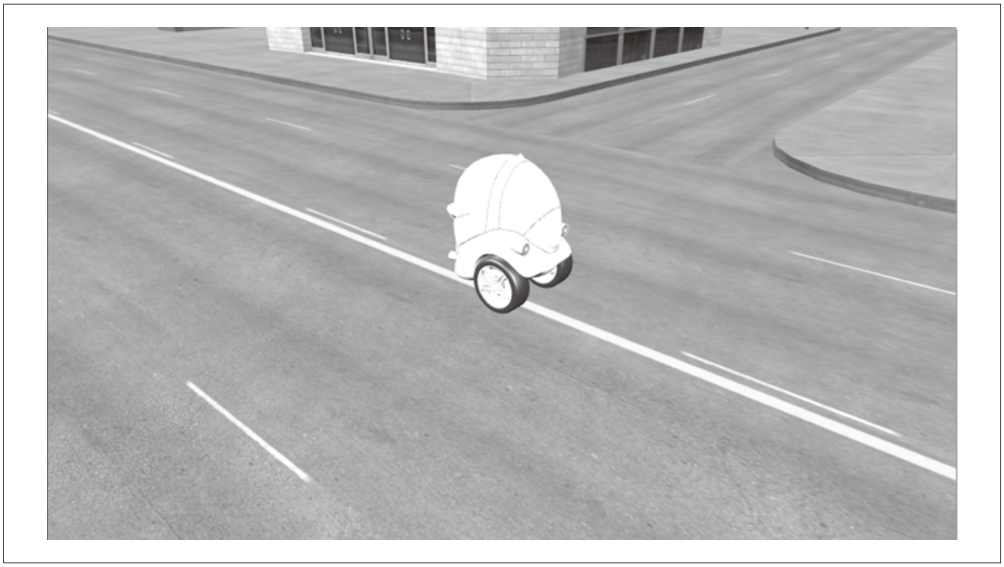


图 11-6：将 Futurgo 模型添加到城市场景中

添加更多模型到已有场景的代码，几乎和加载初始模型的代码一样：创建一个 `Vizi.Loader` 对象，添加一个监听模型加载完成的事件处理函数，并在事件监听函数中添加新的场景对象到查看器中。唯一的不同是我们将对象添加到查看器中，而不是替换它们。例 11-6 展示了相关代码（源文件 Chapter 11/previewer.html）。我们调用 `viewer.addToScene()`，它会添加对象（本例中是 Futurgo 车模型）到当前运行的场景图中，并更新查看器的数据结构。然后我们和之前一样更新用户界面：树状图，以及光源、相机和动画的列表。

例 11-6：插入更多模型到场景中

```
 function onAddComplete(data, loadStartTime)
{
    // 隐藏加载状态栏
    var loadStatus = document.getElementById("loadStatus");
    loadStatus.style.display = 'none';

    viewer.addToScene(data);

    var loadTime = (Date.now() - loadStartTime) / 1000;
    var loadTimeStats = document.getElementById("load_time_stats");
    loadTimeStats.innerHTML = "Load time<br>" + loadTime.toFixed(2) + "s"
    // Vizi.System.log("Loaded " + loadTime.toFixed(2) + " seconds.");

    updateSceneTree(viewer);
    updateCamerasList(viewer);
    updateLightsList(viewer);
    updateAnimationsList(viewer);
    updateMiscControls(viewer);

    addRollovers(viewer, data.scene);
}
```

你或许已经注意到，Futurgo 添加到场景的时候过于明亮了。这是因为 Futurgo 模型包含它自己的光源，我们在第 10 章创建应用的时候使用过。我可以找 TC 做一个没有光源版本的车模型，来在这个应用中使用，但实际上这并不需要。使用预览工具，我们可以找出是哪些光源导致了这个问题，并将它们关掉，然后记下光源的名称，在程序中也这么做。

使用预览工具的光源列表，我们关掉不想要的光源。我注意到它们应该是添加到光源列表后面的光源，因为 Futurgo 模型是最后添加到场景中的。所以我关掉了列表末尾的三个点光源。车依然看起来有点苍白，所以我也关掉了环境光。这下管用了。图 11-7 显示了关掉四个光源后的效果。注意用户界面内椭圆高亮的区域，里面的值改变了。

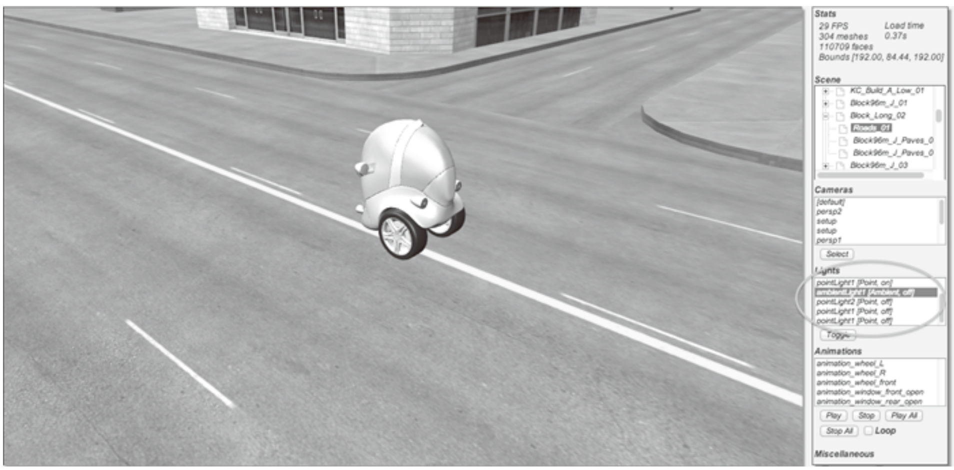


图 11-7：调整光源后城市场景中的 Futurgo 模型

11.2.6　使用预览工具来查找场景中的其他问题

使用预览工具还找到了这个场景中另一个技术问题：树。这个场景的作者使用了久经考验的技巧来低成本渲染树：一系列重叠的平面多边形，上面有从不同角度查看的树纹理贴图。通常，会有两个垂直的多边形以 X 状摆放，再有一个或多个水平的多边形与这个 X 形十字交叉。这是建模师使用多年的一个简单技巧，可以节省多边形数量。否则你设想一下创建逼真的树叶所需的三角形数量吧。

在 WebGL 中使用模型作者关于树的设置时，唯一的问题是图片格式的选择：每个多边形的纹理都是一对微软 BMP 文件，一个是颜色信息，另一个是 alpha 蒙版（alpha mask）。我们不知道如何在 Vizi/Three.js 中轻松地处理它。这在技术上可行，但引擎目前还没有支持。所以我让 TC 将树的一对 BMP 文件转成一个带 alpha 通道的 PNG 文件。他这么做了，然后更新 Maya 文件并重新导出。之前和之后的对比可以查看图 11-8。

虽然右边的图片看起来并没有更好，但它在应用中实际上渲染正确。你这里看到的缺陷是由当前预览工具实现的限制所造成的。尽管在 PNG 文件中有透明度信息，但如果没有在材质上设置透明度值，Vizi 和 Three.js 都无法知道这一信息。因为这些值在模型中没有设置，所以我们需要在应用中等场景加载完后手动设置它们。

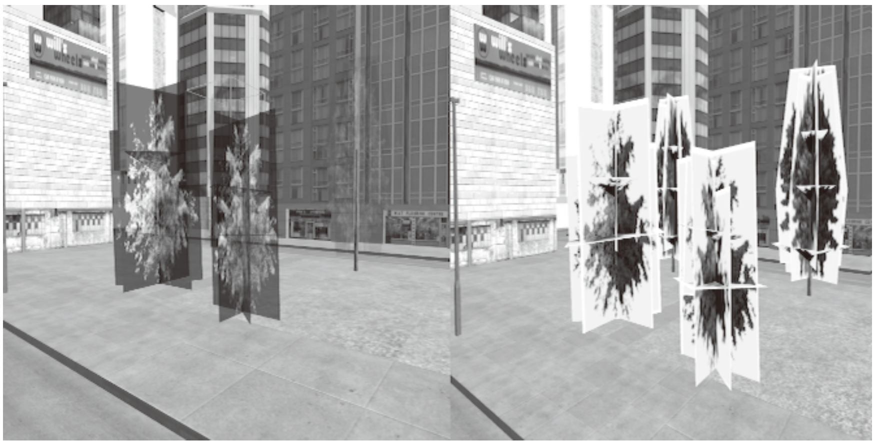


图 11-8：我们并排比较两张图，使用重叠的矩形几何来描述预览工具中的纹理贴图；左边是使用之前两个带 alpha 蒙版的 BMP 文件的效果，右边是使用一个 PNG 文件的效果；右边树周围的白色区域是预览工具的缺陷，当我们在应用代码中明确告诉 Three.js 使用透明度的时候，它就会消失

11.3 使用skybox创建一个3D背景

既然我们预览和调试了城市模型，并且已经知道如何将 Futurgo 与它集成，现在可以开始构建应用了。但首先，我们需要解决另一个问题。这个城市模型的艺术效果确实不错，但它只有有限的 4 个街区。如果我们想要一个迷人的、真实的场景来试驾 Futurgo，就需要创造一个更大城市的假象。我们可以通过渲染一个 skybox 背景来实现。

11.3.1 3D skybox

和典型的 Web 页面背景不同，我们的场景需要 3D 的背景：当相机移动的时候，我们期望看到背景变化。skybox 是一个全景图，由一个立方体内包裹的六个纹理图组成。立方体是通过一个固定相机旋转渲染出来的，会显示出背景的不同部分。skybox 是一种提供真实 3D 背景的极其简单的方法。

Three.js 的示例中包含几个 skybox 功能的演示。打开 Three.js 示例文件 webgl_materials_cubemap_balls_reflection.html 来查看运行的效果。这个例子看起来不错，但是因为 skybox

粗糙的实现，它有一些限制。在所有这些例子中，作者在场景外边缘创建了一个非常大的立方体。它看起来很远，但如果你在场景中导航，能够接近其中某个边缘，甚至最终达到它，从而破坏了假象。



3D 图形中的一切都是假象。skybox 创建了一个令人信服的无限的背景景色假象，只要你永远不通过移动去接近它。如果你看过 Peter Weir 的电影《楚门的世界》，回想一下楚门进入一面实体墙的那个场景。这面墙是节目导演为他绘制的人造世界背景，一旦他撞上那面墙……谎言就被拆穿了。

11.3.2 Vizi skybox对象

为了创建一个令人信服的城市场景，我们需要使用一个合适的 skybox。令人高兴的是，Vizi 框架中自带了一个。在将 skybox 放进城市应用前，我们先创建一个简单的例子来展示它的效果。打开示例文件 Chapter 11/skybox.html，如图 11-9 所示。使用鼠标旋转来查看整个背景，使用触摸板或滚轮来放大缩小。矩形会变近和变远，但背景始终保持无限远的距离。注意 skybox 的背景还反射到了前景立方体的表面。这个效果是通过创建一个具有相同纹理的立方体环境贴图来实现的。

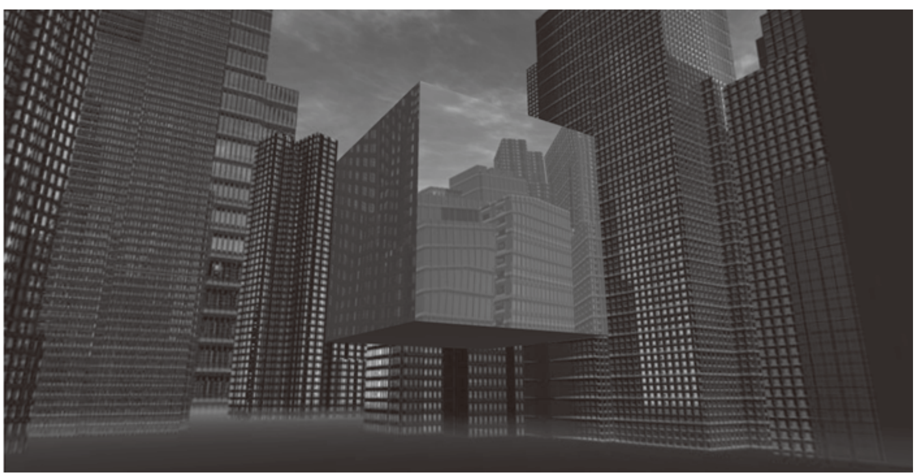


图 11-9：一个 skybox 背景，前景是立方体——当用户向前和向后移动的时候，立方体会变近和变远，但背景保持无限远的距离；立方体上使用同样的纹理贴图来反射背景

skybox 的全景图图片由六个位图组成，它的布局显示在图 11-10 中。这些位图缝合得很紧密，使得映射在立方体内部看起来非常完美。

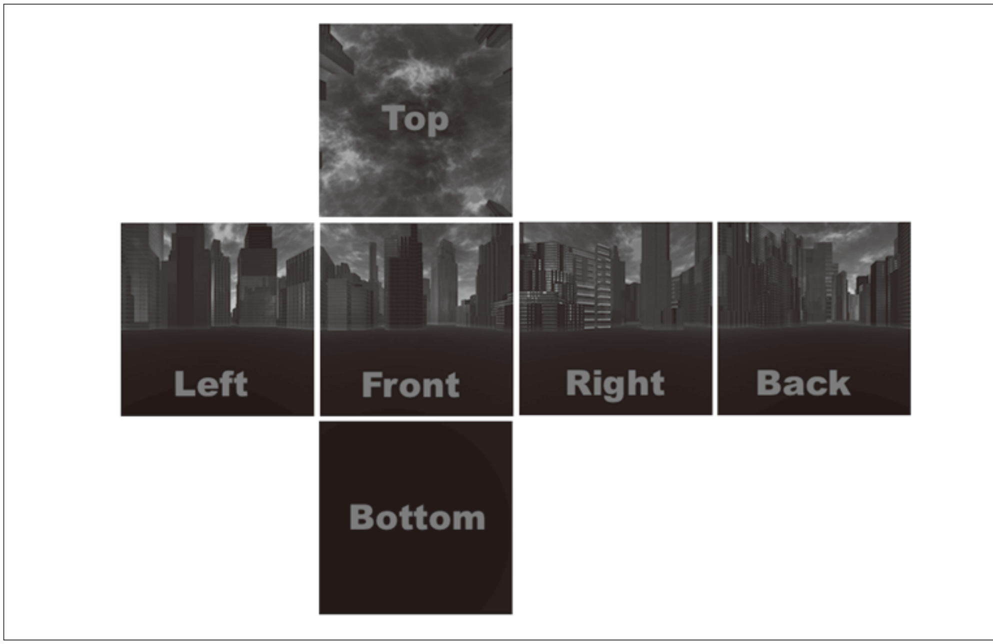


图 11-10：六个纹理组成了 skybox 背景的立方体贴图（skybox 纹理来自 http://www.3delyvisions.com/skf1.htm）

例 11-7 展示了创建 skybox 并添加到场景中的代码。首先，我们使用 Three.js 的辅助工具 `THREE.ImageUtils.loadTextureCube()` 来创建一个立方体纹理贴图。然后调用一个 Vizi 函数来创建 skybox 的 prefab（或者说预建对象）：`Vizi.Prefabs.Skybox()`。最后设置 skybox 的纹理（`texture`）属性为立方体纹理，并将它添加到应用中。

例 11-7：创建天空盒（skybox）背景

```
var app = new Vizi.Application({ container : container });

// 天空盒来自http://www.3delyvisions.com/
// http://www.3delyvisions.com/skf1.htm
var path = "../images/sky35/";

var urls = [ path + "rightcity.jpg", path + "leftcity.jpg",
             path + "topcity.jpg", path + "botcity.jpg",
             path + "frontcity.jpg", path + "backcity.jpg" ];

var cubeTexture = THREE.ImageUtils.loadTextureCube( urls );

var skybox = Vizi.Prefabs.Skybox();
var skyboxScript = skybox.getComponent(Vizi.SkyboxScript);
skyboxScript.texture = cubeTexture;

app.addObject(skybox);
```

那么，这个 prefab 是做什么的呢？

在 Vizi 中，prefab 是一组内置的对象和组件，它可以创建并部署到场景中。prefab 这种设计模式在类似 Unity 这样的游戏引擎中经常出现。现代游戏引擎的设计趋势已经从通过创建类来扩展功能，转向了聚集简单组件到复杂的结构中。

对于 Vizi 的 skybox 来说，它是一个立方体，用来绘制背景，并跟踪主相机的移动，来保持立方体的正确朝向。如果你对 Vizi 中 skybox 的 prefab 是如何实现的感到好奇，可以参考 Vizi 源码中的 objects/skybox.js 文件。

这个例子中的立方体需要反射 skybox 背景图。通过使用相同的立方体纹理贴图来作为立方体的环境贴图，我们很容易就实现了它。具体代码如例 11-8 中所示。

例 11-8：添加立方体贴图到前景对象中

```
var cube = new Vizi.Object;

var visual = new Vizi.Visual(
        { geometry: new THREE.CubeGeometry(2, 2, 2),
            material: new THREE.MeshPhongMaterial({
                color:0xffffff,
                envMap:cubeTexture,
                reflectivity:0.8,
                refractionRatio:0.1
                })
        });


cube.addComponent(visual);
app.addObject(cube);
```

11.4 集成3D内容到应用中

我们已经使用预览工具浏览了场景图，查找到了对象的名称和属性，并查看了光源及其他视觉特性。我们也清楚了当模型加载到应用后需要做的事情。我们还学习了如何构建 skybox 背景，以及在场景中如何反射它到对象上。现在，我们终于准备好集成 3D 环境到应用中了。

11.4.1 加载和初始化场景

Futurgo 试驾应用在文件 Chapter 11/futurgoCity.html 中。其中的 jQuery-ready 代码很简单，它创建了一个会加载模型、组装场景并运行应用的 `FuturgoCity` 类的实例。`FuturgoCity` 的源码可以在文件 Chapter 11/futurgoCity.js 中找到。

应用的设置代码首先加载城市模型。之后文件加载的回调函数 `onLoadComplete()` 开始组装环境。查看例 11-9。当调用查看器的 `replaceScene()` 方法将新加载好的内容添加到场景中后，我们通过调用 `setController("FPS")` 告诉查看器使用第一人称导航。（我们将在本章

后面详细介绍相机控制器及第一人称导航。）然后保存查看器的相机控制器及当前相机的信息，供后面使用。最后，调用几个辅助函数来添加 skybox 和环境贴图，并完成其他重要的设置工作。

例 11-9：环境加载后的回调代码

```
FuturgoCity.prototype.onLoadComplete = function(data, loadStartTime)
{
    var scene = data.scene;
    this.scene = data.scene;
    this.viewer.replaceScene(data);

    if (this.loadCallback) {
        var loadTime = (Date.now() - loadStartTime) / 1000;
        this.loadCallback(loadTime);
    }

    this.viewer.setController("FPS");
    this.cameraController = this.viewer.controllerScript;
    this.walkCamera = this.viewer.defaultCamera;

    this.addBackground();
    this.addCollisionBox();
    this.fixTrees();
    this.setupCamera();
    this.loadFuturgo();
}
```

`addBackground()` 使用与前一节的例子一样的方式创建 skybox，然后添加环境贴图到建筑上。我们已经使用预览工具查找过建筑的名称，它们都以字符串“Tower”或“Office”开头。注意粗体标出的那一行正则表达式。我们使用 Vizi 场景图的 `map()` 方法来查找匹配的对象，然后设置每个对象上 Three.js 材质的环境贴图。

```
        this.scene.map(/Tower.*|Office.*/, function(o) {

            var visuals = o.visuals;
            if (visuals) {
                for (var vi = 0; vi < visuals.length; vi++) {
                    var v = visuals[vi];
                    var material = v.material;
                    if (material) {
                        if (material instanceof THREE.MeshFaceMaterial) {
                            var materials = material.materials;
                            var mi, len = materials.length;
                            for (mi = 0; mi < len; mi++) {
                                addEnvMap(materials[mi]);
                            }
                        }
                        else {
                            addEnvMap(material);
                        }
                    }
                }
            }
        });
```

接下来，我们要添加一个碰撞盒。在本章后面，我们将会看到如何实现碰撞检测，来支持以第一人称模式在场景中导航。现在，这里的代码是在城市的边界设置一个不可见的盒子，使得我们不能超出这些限制。它非常简单：创建一个新的 `Vizi.Visual` 对象，其中包含一个立方体，尺寸为场景的边界框，并通过设置其材质的 `opacity` 属性为 0 来保证它是透明的。除此之外，我们还需要保证碰撞发生在立方体内部。我们通过设置立方体的 `side` 属性为枚举值 `THREE.DoubleSide`（即渲染立方体的两面），来告诉 Three.js 渲染立方体的背面。代码如例 11-10 所示。

例 11-10：添加一个碰撞盒到场景中

```
FuturgoCity.prototype.addCollisionBox = function() {

    var bbox = Vizi.SceneUtils.computeBoundingBox(this.scene);

    var box = new Vizi.Object;
    box.name = "_futurgoCollisionBox";

    var geometry = new THREE.CubeGeometry(bbox.max.x - bbox.min.x,
            bbox.max.y - bbox.min.y,
            bbox.max.z - bbox.min.z);

    var material = new THREE.MeshBasicMaterial({
        transparent:true,
        opacity:0,
        side:THREE.DoubleSide
        });

    var visual = new Vizi.Visual({
        geometry : geometry,
        material : material});

    box.addComponent(visual);

    this.viewer.addObject(box);
}
```

我们需要解决树的透明度问题。在 `fixTrees()` 方法中，我们再次使用 `map()` 来查找名称以“Tree”开头的节点，然后找到它所包含的视觉元素，设置它们材质属性中的 `transparent` 属性为 `true`。这个标记告诉 Three.js 的渲染系统开启 alpha 混合。如果没有它，树就会不透明，如图 11-8 中在预览工具中看到的那样。

```
        this.scene.map(/^Tree.*/, function(o) {

            o.map(Vizi.Visual, function(v){
                var material = v.material;
                if (material instanceof THREE.MeshFaceMaterial) {
                    var materials = material.materials;
                    var i, len = materials.length;
                    for (i = 0; i < len; i++) {
                        material = materials[i];
                        material.transparent = true;
                    }
                }
```

```
            else {
                material.transparent = true;
            }

        });
    });
```

在将相机放到合适的位置来进行初始查看后，我们就可以进行设置应用中最后一个大的步骤：加载车模型。

11.4.2 加载和初始化车模型

加载和准备车模型包含几个步骤：添加加载好的模型到场景中，添加淡出车窗到不同透明度的行为，添加环境贴图到车窗和车身中来反射 skybox，去掉在预览工具中看到的额外光源，最后放置好车。这些都做完后，我们还需要设置开车和相关动画的交互式对象，随后会介绍。

文件加载的回调函数以如下的方式执行。调用 `this.viewer.addToScene()` 来添加对象到场景中。然后，和前一章一样，我们添加车窗的渐变行为，并自动开始，使得它两秒后将变为半透明。另外，我们将渐变的对象保存在应用的 `faders` 属性中。它是一个数组，我们将在后面使用它来渐变车窗，在进入车的时候变得更透明，在离开车的时候变回半透明。我们在遍历车窗材质的时候，还添加了建筑所使用的相同的环境贴图，也就是摩天大楼的 skybox 背景立方体纹理。例 11-11 展示了这些调用。

例 11-11：处理加载 Futurgo 车的回调代码

```
FuturgoCity.prototype.onFuturgoLoadComplete = function(data) {

    // 将Futurgo模型添加到场景中
    this.viewer.addToScene(data);
    var futurgoScene = data.scene;

    // 添加交互和行为
    var that = this;

    // 将环境贴图和淡出器添加到窗口中,在开始阶段将窗口淡出
    this.faders = [];
    futurgoScene.map(/windows_front|windows_rear/, function(o) {

        var fader = new Vizi.FadeBehavior({duration:2,
            opacity:FuturgoCity.OPACITY_SEMI_OPAQUE});
        o.addComponent(fader);
        fader.start();
        that.faders.push(fader);

        var visuals = o.visuals;
        var i, len = visuals.length;
        for (i = 0; i < len; i++) {
            visuals[i].material.envMap = that.envMap;
            visuals[i].material.reflectivity = 0.1;
            visuals[i].material.refractionRatio = 0.1;
        }
```

```
});
```

然后我们将环境贴图添加到车身（金属框架和后视镜）上。

```
// 给车身添加环境贴图
futurgoScene.map(/body2/, function(o) {
    var visuals = o.visuals;
    var i, len = visuals.length;
    for (i = 0; i < len; i++) {
        visuals[i].material.envMap = that.envMap;
        visuals[i].material.reflectivity = 0.1;
        visuals[i].material.refractionRatio = 0.1;
    }

});
```

接下来，我们遍历车身的每一个部分来添加 `Vizi.Picker` 对象。它将允许我们点击 Futurgo 的任意位置来开始试驾。我们将它保存到对象的 `pickers` 数组中，因为我们将会在每次进入和离开车的时候分别禁用和重新开启 picker。

再接下来，我们处理在预览工具中 Futurgo 模型导入到场景后看到的光源问题。Futurgo 模型中额外的光源和场景中的光源叠加，导致模型看起来褪色了。所以我们需要关掉 Futurgo 模型中自带的所有光源。另外，还需要关闭城市模型中提供的环境光。

```
 // 来自两个场景的光同时作用
// 使得车模型看起来过于苍白
// 关掉车模型中自带的所有光源
futurgoScene.map(Vizi.PointLight, function(light) {
    light.intensity = 0;
});

// 还需要关闭城市模型中提供的环境光
this.scene.map(/ambient/, function(o) {
    o.light.color.set(0, 0, 0);
});
```

最后，我们将车放置到一个合适的初始位置，作为我们进入场景时候的视图。因为相机的 *x* 和 *z* 位置值都是零，所以我们将车放在它的右后方。

```
 // 将车放置到一个合适的初始位置
var futurgo = futurgoScene.findNode("vizi_mobile");
futurgo.transform.position.set(2.33, 0, -6);
```

我们还需要添加一些行为和交互来开车，本章末尾部分将对此进行介绍。不过话说回来，现在场景已经完全组装好了。你可以看到 skybox 背景及环境贴图的反射；车已经放好位置了，它的车窗半透明，而且背景的环境贴图反射到车身上。页面加载完后的进入视图如图 11-11 所示。

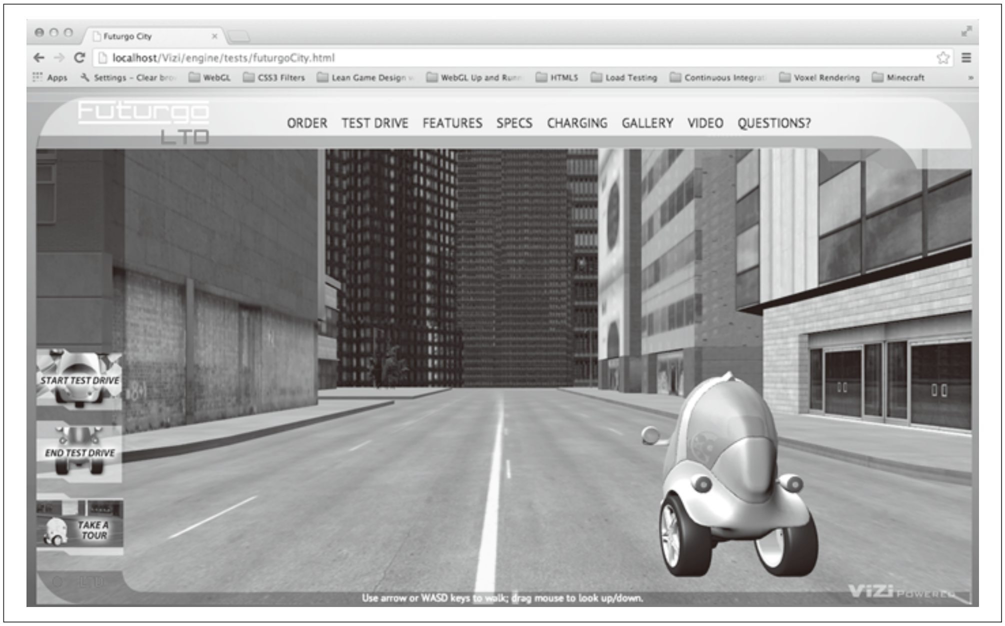


图 11-11：Futurgo 城市应用的进入视图

让我们花些时间来观察。拖拽鼠标来查看四周，使用方向键来在场景中移动。看看建筑高耸而立，反射着黄昏的天空。我们只用了几天就创建好它了，这可真是了不起的成就。

好，闲逛结束了，我们继续工作。

11.5 实现第一人称导航

现在我们加载好环境了，我们需要在其中移动。我们想要允许用户步行浏览城市，或者试驾 Futurgo。在本节中，我们将讨论如何实现游戏风格的导航，也称为*第一人称导航*（first-person navigation）。

第一人称（或第一人称视角）这个术语，指的是从用户的视角来渲染 3D 场景。本质上来说就是将相机放在好似用户双眼间的位置。*第一人称导航*是一种移动相机的模式，相机根据鼠标、键盘、手柄等游戏输入设备进行响应。第一人称导航在电子游戏中非常流行，尤其是*第一人称射击*（First-Person Shooter，FPS）这样的对战游戏。

在桌面电脑上，第一人称导航通常由鼠标和键盘操作，鼠标用于控制相机指向的方向，键盘用于用户的移动、滑行或旋转。表 11-1 展示了第一人称导航中典型的键盘和鼠标绑定。方向键用于视图的左右移动及前进后退，还有 W、A、S 和 D 键（统称“WASD”或“wazz-dee”键）也实现同样的功能，用户能左手进行移动，同时右手使用鼠标旋转相机（或者在射击游戏中射击敌人）。

表11-1：第一人称模式的典型键盘和鼠标绑定

键盘/鼠标行为	功　　能
W、上箭头	前进
A、左箭头	向左移动
S、下箭头	后退
D、右箭头	向后移动
鼠标拖上	相机向上
鼠标拖下	相机向下
鼠标拖左	相机向左
鼠标拖右	相机向右

使用方向键或 WASD 键在城市场景中浏览，拖拽鼠标左键来查看上下左右，如图 11-12 所示。

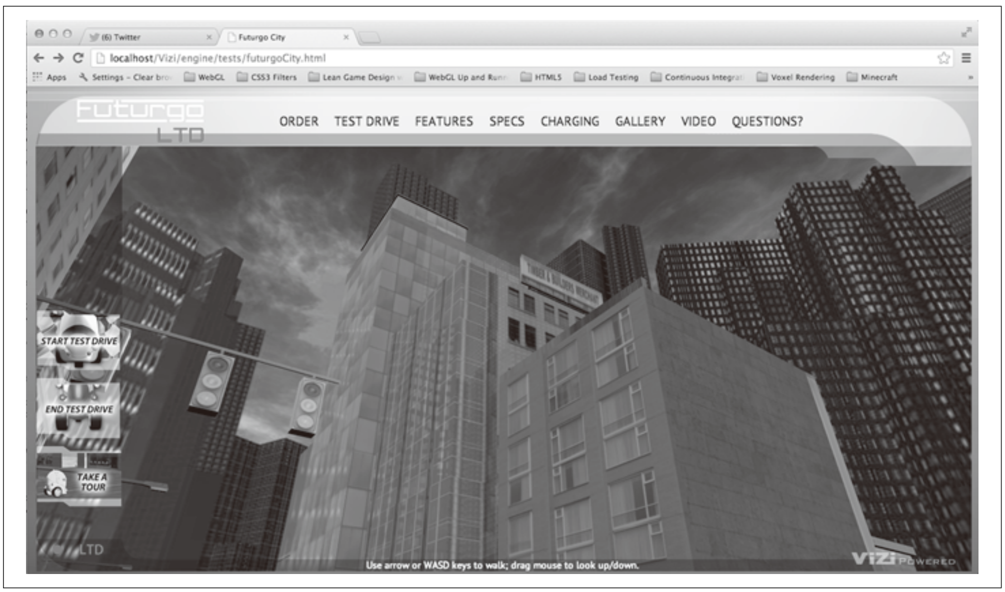


图 11-12：以第一人称模式浏览场景

11.5.1　相机控制器

为了实现第一人称导航，我们将使用相机控制器（camera controller）对象。相机控制器，正如其名，基于用户输入来控制相机的移动。`Vizi.Viewer` 支持不同的相机控制器模式：模型（model）及第一人称（first-person），它会自动为每个模式创建相机控制器对象。只需要调用查看器的 `setController()` 方法，它接收一个字符串来标识使用哪个控制器；目前支持的值是 `"model"` 和 `"FPS"`。

第 10 章中的 Futurgo 应用使用了模型相机控制器，设计用于围绕物体旋转相机，一直朝向

物体的中心。它的效果是当你拖拽鼠标的时候，模型看起会跟着旋转，或者当你使用触摸板或滚轮的时候，模型会变得近点或远点（事实上只有相机在动）。这种类型的相机控制器最适合只使用一个模型的应用。对于城市应用，我们将使用第一人称控制器。

11.5.2 第一人称控制器中的数学

实现第一人称控制器的关键是将鼠标的位置变化转换成相机的旋转：向左向右拖拽会让相机绕着 y 轴旋转，向上向下拖拽则绕着它的 x 轴旋转。基于键盘的移动通常会遵照相机的朝向：按下向上箭头键，相机会沿着视线向前移动。

为了切身体会开发第一人称控制器所需的数学，让我们看看 Vizi 实现中的部分代码。`Vizi.FirstPersonControls` 中的 `update()` 方法会在每次运行循环中被调用，它计算绕 x 及 y 轴旋转的总量。请看例 11-12 中所列出的代码。

例 11-12：`Vizi.FirstPersonControls` 的代码

```
if (this.mouseDragOn || this.mouseLook) {

    var deltax = this.lastMouseX - this.mouseX;
    var dlon = deltax / this.viewHalfX * 900;
    this.lon += dlon * this.lookSpeed;

    var deltay = this.lastMouseY - this.mouseY;
    var dlat = deltay / this.viewHalfY * 900;
    this.lat += dlat * this.lookSpeed;

    this.theta = THREE.Math.degToRad( this.lon );

    this.lat = Math.max( - 85, Math.min( 85, this.lat ) );
    this.phi = THREE.Math.degToRad( this.lat );

    var targetPosition = this.target,
        position = this.object.position;

    targetPosition.x = position.x - Math.sin( this.theta );
    targetPosition.y = position.y + Math.sin( this.phi );
    targetPosition.z = position.z - Math.cos( this.theta );

    this.object.lookAt( targetPosition );

    this.lastMouseX = this.mouseX;
    this.lastMouseY = this.mouseY;
}
```

首先，我们计算鼠标 x 和 y 位置相对于之前的变化，然后将它转换成旋转增量（rotational delta），作为经度和纬度的变化度。局部变量 `dlon` 用于表示经度的变化，我们使用如下公式来计算它：

```
var dlon = deltax / this.viewHalfX * 900;
```

我们用鼠标 x 位置的变化除以场景宽度的一半，来得到鼠标移动的大小占屏幕的百分比。每 10% 的屏幕宽度等于 90 度旋转（因此乘以 900）。然后，我们将这个增量加到当前经度

（水平）的旋转上：

```
this.lon += dlon * this.lookSpeed;
```

经度旋转的角度然后转换成 Three.js 中使用的弧度（radian），并保存到 `this.theta` 属性中：

```
this.theta = THREE.Math.degToRad( this.lon );
```

我们以相同的方式使用鼠标 *y* 位置的变化计算纬度（垂直旋转）的旋转，并保存到 `this.phi` 中。现在我们有了新的经纬度值，可以旋转视图了。我们通过计算出“查看”的位置来实现，这个位置是在以相机为原点的一个单位球体上，然后使用相机的 `lookAt()` 方法让 Three.js 的相机朝向那里。现在，相机已经朝着一个新的方向了。

```
targetPosition.x = position.x - Math.sin( this.theta );
targetPosition.y = position.y + Math.sin( this.phi );
targetPosition.z = position.z - Math.cos( this.theta );
this.object.lookAt( targetPosition );
```

相机随着视线进行移动。如果用户按下任意导航键，我们设置相应的布尔值属性 `moveForward`、`moveBackward`、`moveLeft` 和 `moveRight` 来标识它，并在 `update()` 中进行检查。

```
this.update = function( delta ) {

        this.startY = this.object.position.y;

        var actualMoveSpeed = delta * this.movementSpeed;

        if ( this.moveForward )
            this.object.translateZ( - actualMoveSpeed );
        if ( this.moveBackward )
            this.object.translateZ( actualMoveSpeed );

        if ( this.moveLeft )
            this.object.translateX( - actualMoveSpeed );
        if ( this.moveRight )
            this.object.translateX( actualMoveSpeed );

        this.object.position.y = this.startY;
```

我们使用 Three.js 来帮助我们计算相机的新位置。`translateZ()` 和 `translateX()` 方法分别绕相应的轴进行移动。因为相机或许会在水平方向指向上或下，这将导致沿着 *y* 轴上移动。我们不希望这样，我们希望一直保持在地上。所以我们使用之前保存的值，来覆盖 *y* 位置上的任何修改。

11.5.3 鼠标视角

在这个应用中，用户需要点击和拖拽鼠标来旋转相机视图。相机控制器在许多第一人称游戏中会随着鼠标的移动来旋转视图，而不需要点击。这个模式通常被称为鼠标视角（mouse look）。它对于全屏第一人称游戏来说非常方便，因为它更快和更省力。它还腾出

了鼠标点击 / 放开行为，可以用于射击或打开库存页（inventory page）。

但是对于在窗口中的 Web 导航，鼠标视角则是一个灾难。用户可能想移动鼠标去点击浏览器的地址栏或标签，或者点击页面 2D 界面的元素。但所有这些尝试都会在 3D 窗口中旋转相机视图，使得在用户试图做任何事情的时候相机都会“飞来飞去”，这并不好玩。如果你想尝试一下，可以在这个应用中设置控制器的 `mouseLook` 属性为 `true`，感受一下有多么令人沮丧。在我看来，鼠标视图只适合全屏使用。

> 鼠标视图还可以像许多第一人称游戏中那样隐藏鼠标指针。新的浏览器也支持这个功能，被称为指针锁定（pointer lock）API 和鼠标锁定（mouse lock）API。这个功能的官方 W3C 推荐规范可以在网上找到（https://dvcs.w3.org/hg/pointerlock/raw-file/tip/index.html）。我还推荐一篇由 Google 工程师 John McCutcheon 写的关于这个话题的优秀文章（http://www.html5rocks.com/en/tutorials/pointerlock/intro/）。

11.5.4　简单碰撞检测

一个维持真实环境幻觉的重要特性是*碰撞检测*：检测用户视图（或任意其他对象）碰撞到场景中的几何体，并防止对象穿过几何体。如果用户可以穿过墙，就不能做到令人信服的虚拟城市了。

在本节中，我们将会学习如何实现一个非常简单的碰撞检测，用于 Futurgo 城市环境中。它使用 Three.js math 对象来从视点投射出一条光线，查找在视线上的任意对象。如果在某一特定距离内有任意对象存在，就认为存在碰撞，并禁止我们在那个方向上移动。

类 `Vizi.FirstPersonControllerScript` 是实现 Vizi 第一人称导航系统的 prefab 组件。例 11-13 展示了部分代码。首先，保存原先的相机位置。然后，让 `Vizi.FirstPersonControllerScript` 根据鼠标和键盘输入更新相机位置。接着，调用辅助方法 `testCollision()` 来检测在已保存的位置和新位置之间的移动是否会产生碰撞。如果有碰撞，我们将相机恢复到之前的位置，并触发一个 `"collide"` 事件给监听的函数。（相信我，有函数会监听，稍后会介绍。）

例 11-13：第一人称控制器脚本中的碰撞检测代码

```
Vizi.FirstPersonControllerScript.prototype.update = function()
{
    this.saveCamera();
    this.controls.update(this.clock.getDelta());
    var collide = this.testCollision();
    if (collide && collide.object) {
        this.restoreCamera();
        this.dispatchEvent("collide", collide);
    }
```

现在让我们来看看 `testCollision()` 方法。回忆一下第 9 章介绍过的 `Vizi.Picker` 选取代码。Vizi 图形系统使用 Three.js 的光线投射来查找从视点到几何体之间的相交点。如果有一个光线线段在最小距离 1 及最大距离 2 内，就表明有对象相交，相应的一个对象将被返

回，并保存到 collide 变量中。

```
Vizi.FirstPersonControllerScript.prototype.testCollision = function() {

    this.movementVector.copy(this._camera.position).sub(this.savedCameraPos);
    if (this.movementVector.length()) {

        var collide = Vizi.Graphics.instance.objectFromRay(null,
                this.savedCameraPos,
                this.movementVector, 1, 2);

        if (collide && collide.object) {
            var dist = this.savedCameraPos.distanceTo(collide.hitPointWorld);
        }

        return collide;
    }

    return null;
}
```

以上的算法是最简单的碰撞检测方法。我们使用相机位置来在查看的方向上投射一条光线。因为它是一条光线，没有体积，无限薄，所以这不够真实。真实的化身（avatar）会有曲线或者至少有体积。一个更严格的实现需要分别检测立方体与球体、圆柱体或其他几何体的碰撞。这正是大多数引擎的做法，但对于我们的目的，基于光线的碰撞检测足以避免我们穿过墙壁。

11.6 使用多个相机

3D 的一个好处是可以使用多个相机，因而我们可以使用不同的视角（viewing angle）和高宽比（aspect ratio），从不同的观察点来渲染场景。我们可以一直使用一个相机来完成这一工作，并在需要的时候动态修改它的属性。但是，Three.js 可以轻松创建多个相机，并让它们随时待命。Vizi 框架将 Three.js 的相机封装进了组件，它还管理着对它们的切换，以及处理其他底层的工作，比如当渲染窗口大小改变时自动更新高宽比。我们将在 Futurgo 城市体验中利用这些功能，创建第二个相机并放置在车内。图 11-13 展示了使用这个相机从 Futurgo 内部进行观察的视图。



图 11-13：从 Futurgo 汽车内部的相机查看场景的视图

例 11-14 展示了创建第二个相机的代码。首先，我们创建一个新的 Vizi 对象 `driveCam`，用于包含相机组件。`driveCam` 将会被添加为 Futurgo 车的子节点。为什么这样做？它的好处是当车移动（我们将在后面介绍）的时候，相机会跟着移动。回忆一下我们在之前章节中对变换层级的讨论：一个对象的变换属性（位置、旋转及缩放）会影响它子节点的变换。当车移动或转动的时候，相机会跟着变换。

接下来，我们将相机放到车内。将相机添加为 Futurgo 的子节点会默认放置在车的原点。在这个例子中，相机会在地上。所以我们需要将它放置合适。不过，我们还需要处理一些琐事，因为当 TC 构建 Futurgo 模型时，内部包含了缩放值。（我通过加载模型到预览工具中，并检查顶层组的缩放值进行了确认。）我没有让 TC 重新改变模型尺寸，而是简单地通过除以每个维度的缩放值，来调整相机的位置。结果相机被放置在了大概六英尺（约 1.8 米）高司机坐下后视平线（eye level）的位置，如图 11-13 所示。

例 11-14：创建开车相机

```
// 在车内放置一个相机
var driveCam = new Vizi.Object;
var camera = new Vizi.PerspectiveCamera;
camera.near = 0.01;
driveCam.addComponent(camera);
futurgo.addChild(driveCam);
// 计算模型的拉伸值
// 以便我们将相机摆放在合适的位置
var scaley = futurgo.transform.scale.y;
var scalez = futurgo.transform.scale.z;
var camy = FuturgoCity.AVATAR_HEIGHT_SEATED / scaley;
var camz = 0 / scalez;
driveCam.transform.position.set(0, camy, camz);
this.driveCamera = camera;
```

在下一节中，我将介绍如何切换到这个相机，作为在开始和结束试驾模式的时候进入和离开 Futurgo 车的过渡动画的一部分。

11.7　创建定时的动画过渡

我们快要能够开始试驾了。使用鼠标点击 Futurgo，或点击左边的“Start Test Drive”标签，会让我们进入驾驶室，从而可以开始驾驶。为了让进入车内的过程变得有趣并有几分真实的体验，我们将组合使用 Vizi 组件及基于 `setTimeout()` 的计时器来编写一系列过渡和动画。

例 11-15 中的代码以及后续的代码片段，按照如下步骤实现功能。

(1) 禁用从车内进行选取。我们不希望点击鼠标触发不想要的动画效果。
(2) 通过车窗打开动画来打开驾驶室。
(3) 当车窗打开动画结束后，进入车内。
(4) 一旦进入车内，关闭车窗，并将它们渐变为全透明，使得我们可以看到外面。同时，减弱城市背景音。最后，启用车辆驾驶脚本。

在执行完前面的操作后，我们将会在车内并准备开始驾驶了。下面我们来具体学习每个步骤的代码。

首先，我们禁用选取，并开始车窗打开动画。

例 11-15：进入车内和开始试驾的过渡动画

```
FuturgoCity.prototype.startTestDrive = function(event) {

    if (this.testDriveRunning)
        return;

    this.testDriveRunning = true;

    // 在车体内部禁用选择器
    var i, len = this.pickers.length;
    for (i = 0; i < len; i++) {
        this.pickers[i].enabled = false;
    }

    // 打开车窗
    this.playOpenAnimations();
```

在一秒延迟后，我们进行下一步操作：切换到 `driveCamera` 视图。为此需要设置相机的 `active` 属性为 `true`（在内部实现中，会告诉 Vizi 使用这个新的相机来渲染）。我们还通过设置 `move` 属性为 `false` 来禁用第一人称相机控制器移动的能力。我们还想查看四周，所以继续使用第一人称控制器：在车内，我们将能使用鼠标来倾斜和转动相机方向。

```
        // 在打开车厢后切换到车内相机
        // 并在一段短暂的延迟后开启控制器以供试驾
        var that = this;
        setTimeout(function() {

            // 切换到车内相机
```

```
        that.cameraController.camera = that.driveCamera;
        // 我们需要停留在车内,因此不要移动相机
        that.cameraController.move = false;
        that.driveCamera.rotation.set(0, 0, 0);
        that.driveCamera.active = true;

    }, 1000);
```

现在我们坐在车内，看着外面的世界。接下来让我们在一秒后触发一个过渡序列来关上车窗。我们播放动画来关闭车窗。我们还调小车外的声音（本章后面会讨论如何添加声音到应用中）。我们将车窗渐变到几乎完全透明，以便在驾驶室内可以看到外面。最后，激活驾驶车辆和仪表盘的动画脚本。现在我们准备好试驾了。

```
    // 现在我们在车里,启用车控制器
    // 同时关闭车窗并将其淡出至接近透明
    // 使得我们可以看到车窗外的城市
    setTimeout(function() {

        // 关闭车窗
        that.playCloseAnimations();

        // 实现城市背景音的隔音效果
        that.sound.interior();

        // 全部车窗淡出
        var i, len = that.faders.length;
        for (i = 0; i < len; i++) {
            var fader = that.faders[i];
            fader.opacity = FuturgoCity.OPACITY_MOSTLY_TRANSPARENT;
            fader.start();
        }

        // 启用车脚本控制器和仪表盘动画
        that.carController.enabled = true;
        that.dashboardScript.enabled = true;


    }, 2000);
```

退出试驾模式的代码在 `endTestDrive()` 方法中，在这里我们并没有进行展示。本质上它是反向进行前面开始试驾的步骤。

(1) 禁用车脚本。
(2) 打开车窗。
(3) 重新开启选取；将相机切换至外部视图上；重新激活相机控制器的移动模式；恢复外部声音为正常音量；将车窗渐变回半透明。
(4) 关闭车窗。

11.8 对象行为脚本

现在是时候让车移动了。为此，我们需要编写一个控制器，它类似于在场景中浏览的第

一人称控制器。在这里，键盘移动和选择的是小车而不是相机。（我们希望能用鼠标倾斜和旋转视图，所以继续使用现有的相机控制器，但我们将给它连上内部相机 `driveCamera`，这在前面已经介绍过。）为了创建控制器，我们将使用 Vizi 框架构建一个自定义组件。

11.8.1　基于Vizi.Script实现自定义组件

从根本上来说，Vizi 通过将两个简单的想法相结合来实现它的功能：(1) 一系列控制通用 3D 设计模式的代码（比如开始 / 停止一个行为，查找鼠标下的对象，切换相机）；(2) 能将对象放在一起并让它们进行交互。Vizi 组件几乎对任意对象都有效，因为对象的组织方式是一致的，并且遵循一些简单的规则。例如，每个 `Vizi.Object` 实例都包含一个变换组件，里面有 `position`、`rotation` 和 `scale` 属性，从而允许对象的其他组件控制它。

本章之前讨论的 prefab，比如 `Vizi.Prefabs.Skybox()` 和 `Vizi.Prefabs.FirstPersonController()`，它们作为函数创建了一个预建对象层级结构，并返回一个 `Vizi.Object` 对象作为新创建层级结构的根节点。对象可以是一个单一简单的物体，比如只包含一个立方体；另一方面，它还可以是一个复杂的层级结构，包含几个对象及组件。不仅仅包含简单几何体的 prefab，一般都会有一个或多个脚本来实现这个 prefab 的逻辑。例如 Vizi skybox prefab 包含一个立方体，还有一个脚本用于匹配 skybox 的方向到场景中主相机的方向上。

对于 Futurgo 城市应用来说，我们需要创建一个脚本来驾驶车。如果我们将这个脚本作为一个 Vizi 组件添加到 Futurgo 车对象上，Vizi 框架就会保证每次运行循环的时候调用它的 `update()` 方法，使得它能够响应用户输入，并据此移动车。下面让我们来看看如何实现它。

11.8.2　驾驶车的控制器脚本

回忆一下在模型加载完后初始化车的代码。它添加了选取组件以及淡入淡出等，并添加了环境贴图。它还做了以下工作：

```
// 添加车控制器
this.carController = new FuturgoController({enabled:false,
    scene: this.scene});
futurgo.addComponent(this.carController);
```

创建 `FuturgoController` 组件只是用来完成一件事情：使用方向键来驾驶车。向上箭头加速向前；向下箭头刹车；左箭头和右箭头朝各自方向转弯。这个控制器还测试碰撞，保证车不会穿墙而过。它还遵循地形，当车开上人行道时，会随着路面的升高而升高，而不是“犁”过。因为我们将 `driveCamera` 相机放置在了车内，借助变换层级结构，相机会随着车的移动而移动，所以我们可以享受驾驶。

让我们看看实现这个控制器的代码（例 11-16），源码在文件 Chapter 11/futurgoController.js 中。这个构造函数首先声明自身为 `Vizi.Script` 的子类，`Vizi.Script` 是框架中所有脚本组件的基类。然后它初始化几个属性：移动按键的状态；当前的速度和加速度；一些用来辅助支持碰撞及遵循地形算法的控制变量；以及几个用于帮助实现伪物理算法的时间戳，我们将用这些伪物理算法来控制车的速度。

例 11-16：FuturgoController 组件的构造函数

```
FuturgoController = function(param)
{
    param = param || {};

    Vizi.Script.call(this, param);

    this.enabled = (param.enabled !== undefined) ? param.enabled : true;
    this.scene = param.scene || null;

    this.turnSpeed = Math.PI / 2; // 90 degs/sec

    this.moveForward = false;
    this.moveBackward = false;
    this.turnLeft = false;
    this.turnRight = false;

    this.accelerate = false;
    this.brake = false;
    this.acceleration = 0;
    this.braking = 0;
    this.speed = 0;
    this.rpm = 0;

    this.eyePosition = new THREE.Vector3;
    this.downVector = new THREE.Vector3(0, -1, 0);
    this.groundY = 0;
    this.avatarHeight = FuturgoCity.AVATAR_HEIGHT_SEATED;

    this.savedPos = new THREE.Vector3;
    this.movementVector = new THREE.Vector3;

    this.lastUpdateTime = Date.now();
    this.accelerateStartTime = this.brakeStartTime =
        this.accelerateEndTime = this.brakeEndTime =
        this.lastUpdateTime;
}
```

Vizi 的组件通常会实现两个方法：realize() 和 update()。当渲染、输入、网络或其他浏览器事件需要创建相关数据结构的时候，框架就会调用 realize()。对于车控制器来说，realize() 方法做了两件事情：保存车的初始位置值；创建一个在车发生碰撞的时候触发的反弹行为（bounce behavior）。车的对象可以通过 this._object 属性来获得，Vizi 会在一个组件添加到对象上的时候自动设置好它。

```
FuturgoController.prototype.realize = function()
{
    this.lastUpdateTime = Date.now();

    // 保存地板位置
    this.groundY = this._object.transform.position.y;

    // 添加车发生碰撞时触发的反弹行为
    this.bouncer = new Vizi.BounceBehavior(
```

```
            { duration : FuturgoController.BOUNCE_DURATION }
            );
        this._object.addComponent(this.bouncer);
    }
```

现在来看看 update() 方法：每个对象中的每个组件的 update() 方法，都会在应用的每次运行循环中被调用。对于车控制器，update() 需要做几件事情。首先，它保存车的当前位置，这将会在发生碰撞或者需要根据地形的高低上下移动的时候恢复。然后，它根据内部的物理算法更新速度。接着，它使用速度属性来计算新的位置。最后测试碰撞及地形跟随（terrain following）。

```
    FuturgoController.prototype.update = function()
    {
        if (!this.enabled)
            return;

        var now = Date.now();
        var deltat = now - this.lastUpdateTime;

        this.savePosition();
        this.updateSpeed(now, deltat);
        this.updatePosition(now, deltat);
        this.testCollision();
        this.testTerrain();

        this.lastUpdateTime = now;
    }
```

更新速度涉及使用简单的伪物理算法来模拟加速度和动量，请看例 11-17。向上箭头按得越久，加速度就越大；向下箭头按得越久，刹车就踩得越久，从而让车慢下来。如果没有键按下，但车已经在动了，还会有一定的动量。经过这些计算后，如果速度或加速度有变化，我们还会触发事件来告诉监听器速度变化了。仪表盘控制器（本章后面介绍）会使用这个信息在刻度盘上改变速度及转速（RPM）的视图。

例 11-17：更新车速

```
FuturgoController.prototype.updateSpeed = function(now, deltat) {

    var speed = this.speed, rpm = this.rpm;

    // 在油门踩下时加速
    if (this.accelerate) {
        var deltaA = now - this.accelerateStartTime;
        this.acceleration = deltaA / 1000 * FuturgoController.ACCELERATION;
    }
    else {
        // 动量
        var deltaA = now - this.accelerateEndTime;
        this.acceleration -= deltaA / 1000 * FuturgoController.INERTIA;
        this.acceleration = Math.max( 0, Math.min( FuturgoController.MAX_ACCELERATION,
            this.acceleration) );
    }
```

```
    speed += this.acceleration;

    // 踩下刹车时减速
    if (this.brake) {
        var deltaB = now - this.brakeStartTime;
        var braking = deltaB / 1000 * FuturgoController.BRAKING;

        speed -= braking;
    }
    else {
        // 惯性
        var inertia = deltat / 1000 * FuturgoController.INERTIA;
        speed -= inertia;
    }

    speed = Math.max( 0, Math.min( FuturgoController.MAX_SPEED, speed ) );
    rpm = Math.max( 0, Math.min( FuturgoController.MAX_ACCELERATION,
        this.acceleration ) );

    if (this.speed != speed) {
        this.speed = speed;
        this.dispatchEvent("speed", speed);
    }

    if (this.rpm != rpm) {
        this.rpm = rpm;
        this.dispatchEvent("rpm", rpm);
    }
}
```

为了改变车的位置，我们需要使用当前的速度来在当前视线（z 轴的负方向）上移动。我们还要让对象围绕自身 y 轴旋转来控制车的转弯。

```
FuturgoController.prototype.updatePosition = function(now, deltat) {

    var actualMoveSpeed = deltat / 1000 * this.speed;
    var actualTurnSpeed = deltat / 1000 * this.turnSpeed;

    // z轴上的运动
    this._object.transform.object.translateZ( -actualMoveSpeed );

    // 使车辆始终在地面上行驶
    this._object.transform.position.y = this.groundY;

    // 转弯
    if ( this.turnLeft ) {
        this._object.transform.object.rotateY( actualTurnSpeed );
    }

    if ( this.turnRight ) {
        this._object.transform.object.rotateY( -actualTurnSpeed );
    }

}
```

1. 检测车与场景的碰撞

检测车与建筑间碰撞的代码和 `Vizi.FirstPersonController` 中所用的碰撞检测代码类似。它调用图形系统的 `objectFromRay()` 方法来计算从当前相机位置到场景中任意几何体的相交。注意传递给 `objectFromRay()` 的第一个参数 `this.scene`，它包含城市场景中的所有几何体，但不包括车本身。如果我们在碰撞检测中包括车的几何体，它将永远返回 `true`。

```
FuturgoController.prototype.testCollision = function() {

    this.movementVector.copy(this._object.transform.position)
      .sub(this.savedPos);
    this.eyePosition.copy(this.savedPos);
    this.eyePosition.y = this.groundY + this.avatarHeight;

    var collide = null;
    if (this.movementVector.length()) {

        collide = Vizi.Graphics.instance.objectFromRay(this.scene,
            this.eyePosition,
                this.movementVector,
                FuturgoController.COLLISION_MIN,
                FuturgoController.COLLISION_MAX);

        if (collide && collide.object) {
            var dist = this.eyePosition.distanceTo(collide.hitPointWorld);
        }
    }

    if (collide && collide.object) {
        this.handleCollision(collide);
    }

}
```

2. 实现碰撞响应

在我们浏览城市的时候，第一人称控制器避免我们穿过实体建筑。当碰撞发生的时候，我们就会停下来。而对于车，我们想要有些不同。在真实世界中，当车撞上建筑的时候，它如果不撞碎就会被反弹。我们的模拟比较随意，在 Futurgo 车碰撞到场景中物体的时候会轻轻反弹。3D 应用响应物体碰撞的概念被称为*碰撞响应*（collision response）。

例 11-18 展示了在车控制器中如何实现反弹碰撞响应。首先，我们触发一个“collide”事件给所有监听器。应用会监听这个事件，在车碰撞的时候触发一个声音。然后，我们调用 `restorePosition()` 方法恢复车的原始位置，来避免它穿过碰撞体。接下来，回忆一下在 `realize()` 方法中，我们向 Futurgo 车添加了一个 `Vizi.BounceBehavior` 组件。我们触发那个反弹行为，它会让车向后反弹一点。这里的向后是指向移动方向的反方向。请看我们如何设置 bouncer 的 `bounceVector` 属性为移动向量的负数，并缩放到 1/3 来模拟部分动量在“碰撞”的时候被吸收了。最后，我们关闭引擎。

例 11-18：一个碰撞响应

```
FuturgoController.prototype.handleCollision = function(collide) {
```

```
    // 通知所有的监听器
    this.dispatchEvent("collide", collide);

    // 后退到上一次保存的位置
    this.restorePosition();

    // 触发反弹行为
    this.bouncer.bounceVector
        .copy(this.movementVector)
        .negate()
        .multiplyScalar(.333);
    this.bouncer.start();

    // 关闭引擎(让车停下)
    this.speed = 0;
    this.rpm = 0;
}
```

3　实现地形跟随

城市环境非常平坦，但还是有些地方会凸起。人行道会比道路要高一点。我们需要决定当 Futurgo 开到上面的时候如何处理，可以选择停止或开上人行道。但我们不希望车“犁”过人行道，好像人行道不存在一样。在碰到人行道的时候，停止是一个简单的解决方法，但这很无趣。因此我们将让车开到人行道上，为此需要实现地形跟随。

*地形跟随*是指使相机或化身与地面保持固定距离的算法。当相机在场景中移动的时候，会向下投射一束光线。如果它碰上任何几何体，将会检查几何体的距离，和相机的期望高度进行比较：如果距离小于期望值，相机就会上升，看起来升高了；如果距离比期望值大，相机就会下移。

Futurgo 车控制器在它的 update 方法中每次都会执行地形跟随检测。我们再一次使用 Vizi 的方法来测试和场景几何体的碰撞，不过这一次是向下的光线（`downVector = [0, -1, 0]`）。你可以自己测试一下，驶向建筑，车会开上人行道。开回街道的时候，车会落到路面。查看例 11-19 中的代码，以及图 11-14 中碰撞及地形跟随的演示。

例 11-19：Futurgo 中的地形跟随

```
FuturgoController.prototype.testTerrain = function() {

    var EPSILON = 0.00001;

    var terrainHit = Vizi.Graphics.instance.objectFromRay(this.scene,
              this.eyePosition,
              this.downVector);

    if (terrainHit && terrainHit.object) {
        var dist = this.eyePosition.distanceTo(terrainHit.hitPointWorld);
        var diff = this.avatarHeight - dist;
        if (Math.abs(diff) > EPSILON) {
            console.log("distance", dist);


            this.eyePosition.y += diff;
```

```
            this._object.transform.position.y += diff;
            this.groundY = this._object.transform.position.y;
        }
    }
}
```

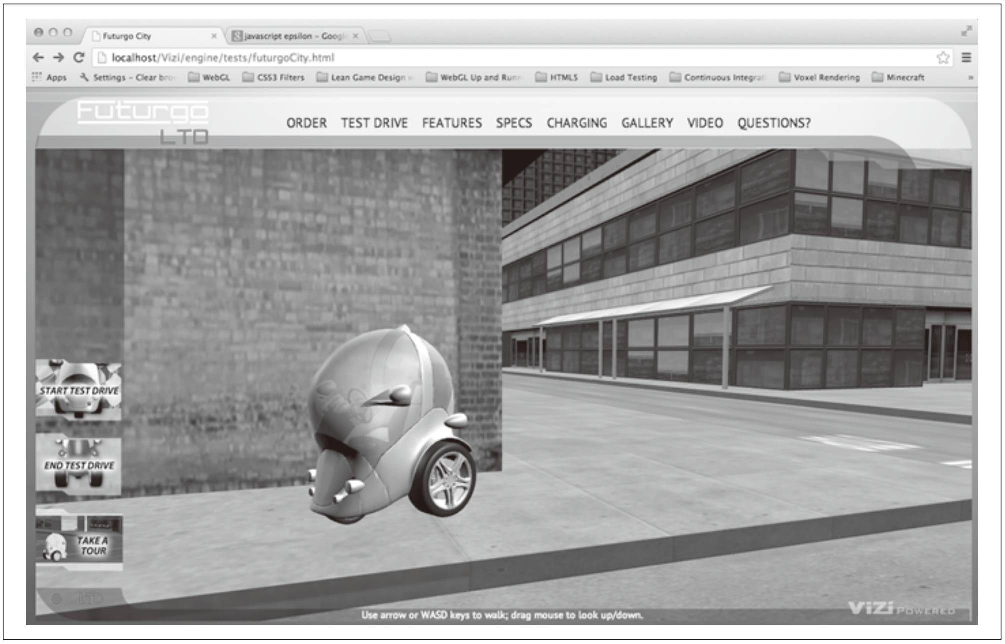


图 11-14：碰撞和地形跟随——车停在墙边，它使用光线投射来开到人行道上

11.9　给环境添加声音

现在已经非常有趣了，我们可以驾驶着 Futurgo 在城市道路中自由行驶，但还缺了一样东西：声音。声音对于基于页面的 Web 应用来说有些奢侈，但在一个真实 3D 场景中，缺少它会很显眼。谢天谢地，使用标准 HTML5 音频可以轻松添加基本声音。

对于这个应用，我们只需要两个声音：一个循环播放的城市背景环境声，一个车碰撞到物体时所发出的“撞击”声。首先，我们添加 <audio> 和 <sound> 元素到 HTML 页面中（文件 Chapter 11/futurgoCity.html）：

```
<audio volume="0.0" id="city_sound">
<!-- http://www.freesound.org/people/synthetic-oz/sounds/162704/  -->
  <source src="../sounds/162704__synthetic-oz__city-trimmed-looped.wav"
    type="audio/wav" />
  Your browser does not support WAV files in the audio element.
</audio>
<audio volume="0.0" id="bump_sound">
<!-- http://www.freesound.org/people/Calethos/sounds/31126/  -->
  <source src="../sounds/31126__calethos__bump.wav" type="audio/wav" />
```

```
    Your browser does not support WAV files in the audio element.
  </audio>
```

现在我们只需要写一点代码来改变音量和触发声音播放。当我们进入 Futurgo 进行试驾的时候，城市声音的音量应该低一些；在我们离开驾驶室的时候，城市声音的音量应该恢复原有大小。当我们驾车发生碰撞的时候，需要播放一次撞击声。

声音实现的源码在文件 Chapter 11/futurgoSound.js 中。它非常简单，使用标准 HTML5 DOM audio 方法。例 11-20 展示了全部的代码。`interior()` 和 `exterior()` 方法分别提高和降低环境背景音的音量。`bump()` 方法播放一次撞击声。

例 11-20：在 Futurgo 城市场景中管理声音

```
FuturgoSound = function(param) {

    this.citySound = document.getElementById("city_sound");
    this.citySound.volume = FuturgoSound.CITY_VOLUME;
    this.citySound.loop = true;

    this.bumpSound = document.getElementById("bump_sound");
    this.bumpSound.volume = FuturgoSound.BUMP_VOLUME;
}

FuturgoSound.prototype.start = function() {

    this.citySound.play();

}

FuturgoSound.prototype.bump = function() {

    this.bumpSound.play();

}

FuturgoSound.prototype.interior = function() {

    $(this.citySound).animate(
            {volume: FuturgoSound.CITY_VOLUME_INTERIOR},
            FuturgoSound.FADE_TIME);
}

FuturgoSound.prototype.exterior = function() {

    $(this.citySound).animate(
            {volume: FuturgoSound.CITY_VOLUME},
            FuturgoSound.FADE_TIME);
}

FuturgoSound.prototype.bump = function() {

    this.bumpSound.play();

}
```

```
FuturgoSound.CITY_VOLUME = 0.3;
FuturgoSound.CITY_VOLUME_INTERIOR = 0.15;
FuturgoSound.BUMP_VOLUME = 0.3;
FuturgoSound.FADE_TIME = 1000;
```

唯一剩下的事情就是在应用中调用这些方法。回过头来看看 `startTestDrive()` 中的操作步骤（文件 Chapter 11/futurgoCity.js）：

```
// 创建城市背景音的隔音效果
that.sound.interior();
```

在离开车的时候调用 `exterior()` 方法来恢复背景音到初始音量。

`FuturgoCity` 类还处理了碰撞声音，添加一个事件监听到车控制器中：

```
this.carController.addEventListener("collide", function(collide) {
        that.sound.bump();
    });
```

11.10　渲染动态纹理

我们已经到了创建真实环境之旅的最后一程。Futurgo 车已经可以开动了。在实现声音后，我以为我已经完成了所有代码的编写。但当我进入到车内驾驶的时候，却感觉毫无生机。我很快意识到，这是因为控制面板的刻度表是不动的：当车移动的时候，车速表和转速表上的刻度盘并没有跟着变化。和之前声音的例子一样，环境的真实性提升了我的期望值。当车移动的时候，刻度盘应该跟着旋转。所以我们需要让仪表盘动起来，或者至少让它的纹理贴图动起来。

在本节中，我们将创建一个*程序纹理*（procedural texture），它是通过程序代码动态绘制的纹理贴图（而不是从一个文件加载进来的静态图片）。为此，我们需要使用后备的 2D canvas 渲染。仪表盘使用 2D Canvas API 来生成一个程序纹理，用于表示当前仪表上的速度值和转速值。

Futurgo 仪表盘上最初的纹理贴图有固定位置的刻度指针。我让 TC 将刻度指针从仪表盘中分离出来，作为一个单独的图片。他完成后给了我分离好的图片。TC 并不需要修改 3D 模型，他只需修改纹理。这两个位图文件如图 11-15 和图 11-16 所示。



图 11-15：仪表盘的纹理贴图



图 11-16：可旋转刻度指针的纹理贴图

为了实现仪表盘动画，我们将创建另一个自定义 Vizi 组件 `FuturgoDashboard`。`FuturgoDashboard` 是一个脚本，它创建一个 HTML Canvas 元素，在 `realize()` 方法中加载刚才的两个位图，并在 `update()` 方法中根据当前的车速和转速来更新刻度指针。我们将在 `FuturgoController` 中通过添加事件监听来跟踪车速和转速的变化。

例 11-21 展示了设置它的代码。`realize()` 创建了一个新的 Canvas 元素，使用 Three.js 纹理对象来容纳它。然后我们设置这个新的纹理为仪表盘材质的 `map` 属性。然后，我们可以

使用标准的 Canvas 2D 绘图 API 调用来更新 Canvas 的内容，这些修改会反映到物体的纹理贴图上。

例 11-21：为仪表盘创建一个 canvas 纹理

```
FuturgoDashboardScript.prototype.realize = function()
{
    // 设置仪表盘盘面刻度
    var gauge = this._object.findNode("head_light_L1");
    var visual = gauge.visuals[0];

    // 创建一个新的用于绘制的canvas元素
    var canvas = document.createElement("canvas");
    canvas.width = 512;
    canvas.height = 512;

    // 以这个canvas元素为源创建一个Three.js纹理对象
    var texture = new THREE.Texture(canvas);
    texture.wrapS = texture.wrapT = THREE.RepeatWrapping;
    visual.material.map = texture;

    this.texture = texture;
    this.canvas = canvas;
    this.context = canvas.getContext("2d");
```

继续看 `realize()` 的后半部分，到了加载纹理的代码。我们使用 DOM 属性来实现：首先，设置一个 `onload` 处理函数，它会告诉我们什么时候图片加载完并准备好；然后，我们设置 `src` 属性来加载图片。

```
        // 加载仪表盘和刻度的纹理
        this.dashboardImage = null;
        this.dialImage = null;

        var that = this;
        var image1 = new Image();
        image1.onload = function () {
            that.dashboardImage = image1;
            that.needsUpdate = true;
        }
        image1.src = FuturgoDashboardScript.dashboardURL;

        var image2 = new Image();
        image2.onload = function () {
            that.dialImage = image2;
            that.needsUpdate = true;
        }
        image2.src = FuturgoDashboardScript.dialURL;

        // 强制进行初始化更新
        this.needsUpdate = true;
    }
```

现在可以绘制纹理了，查看例 11-22。每次调用仪表盘 `update()` 方法的时候，我们将根据车控制器的速度值或转速值是否变化，来决定是否需要重绘纹理。如果有变化，我们调用

darw() 方法来应用 Canvas API 绘制到纹理上。首先，draw() 会使用当前文本的颜色清空 canvas 的内容。然后，如果仪表板位图加载好了，它会使用上下文的 drawImage() 方法进行绘制，使用图片的像素来覆盖整个 canvas。

例 11-22：绘制仪表盘背景图像

```
FuturgoDashboardScript.prototype.draw = function()
{
    var context = this.context;
    var canvas = this.canvas;

    context.clearRect(0, 0, canvas.width, canvas.height);
    context.fillStyle = this.backgroundColor;
    context.fillRect(0, 0, canvas.width, canvas.height);

    context.fillStyle = this.textColor;

    if (this.dashboardImage) {
        context.drawImage(this.dashboardImage, 0, 0);
    }
```

如果你对 Canvas API 不熟悉，第 7 章介绍了 Canvas 绘图基础。

现在，我们需要将刻度指针画在上面。我们已经跟踪了车速和转速（待会儿会详细介绍），将使用这些值来计算刻度指针位图旋转的角度。回忆一下 2D Canvas API 中提供的 save() 和 restore() 方法，它们用于在一系列绘制操作前保存上下文的当前状态，并在完成绘制后恢复。我们会在绘制每个刻度指针的时候调用它们。保存好状态后，我们在上下文上执行 2D 变换，将我们即将绘制的刻度指针像素平移到仪表正确的位置上，然后旋转到正确的位置上，和当前车速值和转速值匹配。然后，我们绘制图片并恢复上下文。我们对每个仪表都这么操作。（基于刻度指针位图的尺寸，我得到了这里所用到的平移值；通过一个图片编辑程序，我得到了一个可以确定的位置。）

```
        var speeddeg = this._speed * 10 - 120;
        var speedtheta = THREE.Math.degToRad(speeddeg);
        var rpmdeg = this._rpm * 20 - 90;
        var rpmtheta = THREE.Math.degToRad(rpmdeg);

        if (this.dialImage) {
            context.save();

            context.translate(FuturgoDashboardScript.speedDialLeftOffset,
                    FuturgoDashboardScript.speedDialTopOffset);
            context.rotate(speedtheta);
            context.translate(-FuturgoDashboardScript.dialCenterLeftOffset,
                    -FuturgoDashboardScript.dialCenterTopOffset);
            context.drawImage(this.dialImage, 0, 0);
            context.restore();

            context.save();
```

```
            context.translate(FuturgoDashboardScript.rpmDialLeftOffset,
                    FuturgoDashboardScript.rpmDialTopOffset);
            context.rotate(rpmtheta);
            context.translate(-FuturgoDashboardScript.dialCenterLeftOffset,
                    -FuturgoDashboardScript.dialCenterTopOffset);
            context.drawImage(this.dialImage, 0, 0); // 198, 25, 115);
            context.restore();
        }
    }
```

唯一剩下的事情就是连接车控制器到仪表盘上，使它可以监听车速和转速的变化。

城市应用在仪表盘创建后设置它的 carController 属性：

```
        this.dashboardScript.carController = this.carController;
```

carController 的代码展示在例 11-23 中，它是一个使用 Object.defineProperties 创建的 JavaScript 属性。在内部实现中，设置属性会调用对象的访问方法 setCarController()。该方法保存控制器到一个私有属性 this._carController 中，并添加车控制器“speed”和“rpm”事件的监听器。这些监听器会保存新的值，然后通过设置仪表盘的 needsUpdate 属性来标记它需要重绘。现在，当车速提高或降低的时候，仪表盘的显示界面会进行重绘来反映这一变化。

例 11-23：仪表盘控制器脚本设置车速和转速改变的监听器

```
FuturgoDashboardScript.prototype.setCarController =
  function(controller) {

    this._carController = controller;

    var that = this;
    controller.addEventListener("speed", function(speed) {
      that.setSpeed(speed); });
    controller.addEventListener("rpm", function(rpm) {
      that.setRPM(rpm); });
}
```

使用 Canvas 元素作为 WebGL 纹理的能力非常强大。它允许开发者使用熟悉、简单的 API 来在 JavaScript 中动态绘制纹理，开启了实现激动人心效果的可能。WebGL 的设计者在这一点上是正确的。WebGL 还支持 HTML 视频元素纹理，可以进行更强大的组合。

11.11 小结

这一章很长，但涉及了许多内容。

你学到了如何在 Web 页面中实现一个可用的、看起来真实的 3D 环境，它带有全景背景、环境贴图反射、用户控制的导航、音效设计，以及可以使用交互行为来移动的物体。我们扩展了我们的工具，给预览工具添加了新功能，让它允许我们查看场景图的结构，以及每

个对象的详细属性。你学习了如何开发几个经典 3D 游戏算法和特效的简单版本，比如第一人称导航、碰撞检测、地形跟随、skybox 渲染以及程序纹理。

在浏览器中创建 3D 环境是一项复杂的工作，但它可以在常规 Web 开发的时间和预算内完成。现在，你对如何完成它已经有所了解。

第12章

开发移动3D应用

随着过去 10 年的发展，HTML5 在手机和平板电脑中发生了更具革命性的演变。苹果 iPhone 和 iPad 的普及模糊了移动设备和传统电脑间的界限。移动设备目前在每年的发货量上超过了传统计算机，因为消费者期待更简单、更小巧和更便携的设备来玩游戏、看视频、听音乐、写邮件、上网以及打电话。这些新的掌上设备还提供了大量新功能，包括基于位置的服务、触摸屏界面、设备方向输入。

为了能够使用智能手机和平板电脑所提供的新功能，开发者通常需要学习新的编程语言和操作系统。比如，为苹果设备开发应用需要使用 iOS 系统的 API，并基于 Objective-C 语言（或者从其他类似于 C++ 这样的原生语言桥接到它）进行开发；为 Android 操作系统开发需要学习不同的 API，并基于 Java 开发。一段时间以来，移动平台只为基于 HTML5 开发提供了有限的能力，它使用基于 WebKit 的控件来嵌入到应用中。移动平台允许开发者使用 HTML、CSS 及 JavaScript 开发演示脚本及一些应用逻辑，但为了访问平台的功能，还需要使用原生代码来编写应用的大部分内容，包括基于 OpenGL 的 3D 图形，这些都还没有在移动浏览器中支持。

过去的几年，浏览器追赶上来了。大部分最初在移动平台上创新的功能已经进入 HTML5 规范。针对特定设备的原生移动开发及 Web 开发这两个曾经分开的世界，看起来就要融合了。对于许多 Web 和移动应用开发者而言，这是一件好事：HTML5 和 JavaScript 可以降低开发成本，还有开发出真正跨平台应用的潜力。3D 是最近增加进去的功能。移动设备对 CSS3 的支持很普遍，而 WebGL 现在已经快被移动平台普遍支持了。在本章中，我们将看看开发基于 HTML5 的移动 3D 应用时会遇到的问题。

12.1　移动3D平台

尽管原生 API 还在功能上领先于 HTML5，但距离在快速缩小。尽管存在一些限制，但 3D

已经在大部分移动浏览器中得到支持。大部分浏览器支持 WebGL，但有些，比如移动版 Safari 还不支持。在本书写作之际，以下是在移动设备上开发基于 HTML5 应用的现状。

- 很多移动浏览器支持 WebGL，但并不是全部，表 12-1 汇总了支持 WebGL 的浏览器。
- 所有移动浏览器都支持 CSS 3D 变换、过渡及动画。第 6 章开发的例子应该能在任意移动浏览器环境下运行。如果你的应用的 3D 需求很简单，主要是在 2D 页面元素中的 3D 特效，你应该认真考虑使用 CSS3 而不是 WebGL，因为 WebGL 并没有在移动设备中得到全面支持。
- 所有移动浏览器都支持 2D Canvas API。它可以用来作为不支持 WebGL 的移动平台的潜在降级方案，虽然它有性能问题，因为 2D Canvas 元素并没有硬件加速。

表12-1：移动设备和操作系统中的WebGL支持

平台/设备	支持的浏览器
Amazon Fire OS（基于 Android）	Amazon Silk（只有 Kindle Fire HDX）
Android	移动版 Chrome、移动版 Firefox
Apple iOS	在移动版 Safari 和移动版 Chrome 中不支持；在 iAds 框架中支持，它用来在应用中创建基于 HTML5 的广告
BlackBerry 10	BlackBerry Browser
Firefox OS	移动版 Firefox
Intel Tizen	Tizen Browser
Windows RT	Internet Explorer（需要 Windows RT 8.1 或更高版本）

在上面的表格中最明显的区别就是：在 iOS 上的移动版 Safari 和移动版 Chrome 中缺乏对 WebGL 的支持。尽管 Android 在移动市场份额上取得了巨大的成绩，其他系统也逐渐流行起来，但 iOS 依然是一个非常流行的移动平台，对开发者有着极大的吸引力。iOS 的情况或许未来会改变，但现实是目前 iOS 中的浏览器不支持 WebGL。

在浏览器不支持 WebGL 的平台上，有一些适配技术，被称为“混合”（hybrid）解决方案，它为应用程序提供 WebGL API。开发者可以使用 JavaScript 来编写他们的应用，JavaScript 能调用一系列原生代码实现 API。结果并不是一个基于浏览器的应用，但有原生代码的执行性能，并且可以获得 JavaScript 快速简单开发的好处。我们将在本章后面研究其中一种技术：Ludei 公司的 CocoonJS。

对于支持 WebGL 的移动平台，有两种部署方案：基于浏览器的应用和打包的应用（通常被称为 Web 应用）。对于基于浏览器的移动 WebGL，你只需要像开发桌面平台应用那样进行开发，然后发布到你的服务器上。对于 Web 应用，你需要使用平台的工具来打包文件（这些文件通常是你要发布到服务器上的那些，或许还加入一个图标和一些元数据信息），然后通过平台的应用商店或类似的服务来发布。

不管你如何部署你的应用，有一些你在开发移动 3D 时需要特别注意的地方。首先，你将需要添加对设备能力的支持，比如触摸输入、位置和设备方向。你还需要更加留心性能，因为移动设备上的内存更小并且（通常）CPU 和 GPU 的运算能力更弱。我们将在本章后面介绍这些话题。

HTML5 移动平台在不断变化中，似乎每天都会出现新的平台。维基词条（http://en.wikipedia.org/wiki/HTML5_in_mobile_devices）对相关背景信息进行了恰当的概述。

12.2 为移动浏览器开发

如果你已经有为桌面浏览器创建 WebGL 应用的经验，开始移动开发会很简单，只需将浏览器指向相关 URL。如果你的移动平台支持 WebGL，它应该会正常运行。性能肯定会有差异。设备和操作系统平台支持许多不同的硬件配置：有些非常低端，比如来自 GeeksPhone 的 Firefox OS 手机；其他的性能都还不错，比如新的 Samsung Galaxy 和 Google Nexus 平板。但它们至少都能在屏幕中渲染一些效果，让你在此基础上开发。

在我测试的设备中，让我印象最深刻的就是 Amazon Kindle Fire HDX，它于 2013 年 10 月发布。Kindle Fire HDX 是 Kindle Fire 产品线的升级，有优秀的硬件配置：一颗来自高通的骁龙（Snapdragon）四核处理器，以及加上了对 HTML5 一流支持的 Adreno 330 GPU。这个七英寸的平板对本书中的例子支持得很好。图 12-1 是 Futurgo 概念车网站（参见第 10 章）运行在 Kindle Fire HDX 中的 Amazon Silk 浏览器上的截图。注意，它看起来和第 10 章中桌面版的例子一样。手指在 canvas 区域滑动来旋转 Futurgo，用两个手指在 canvas 区域捏合来缩放模型，点击 Interior 和 LTD Racing 标签来开启动画，点击 Futurgo 车身的一部分来弹出浮层。它的性能很不错，对于一个超级轻的七英寸手持设备来说效果超出预期。

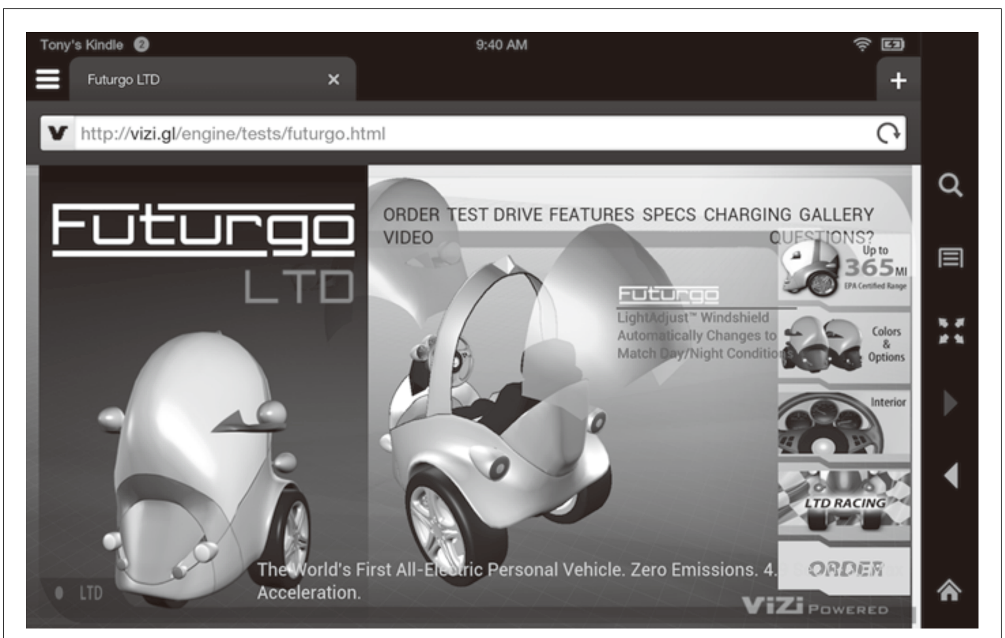


图 12-1：运行在 Kindle Fire HDX 上的 Futurgo 应用

我并没有做任何附加工作来让这个例子在 Kindle 设备上运行，这不足为奇。我只是将 URL 输入到浏览器中，等待几秒，页面就完全渲染并动起来了。像这样的移动设备没有鼠标，所以为了实现交互，我需要添加触摸输入的支持。

12.2.1 增加触摸支持

移动 HTML5 浏览器会自动处理页面元素的触摸输入，生成合适的 `mouseclick` 事件。Futurgo 页面右边的标签可以正常工作，能够触发打开车和旋转轮子的动画。但是，浏览器不会自动为 Canvas 元素生成鼠标事件，我们需要自己通过处理浏览器触摸事件（touch event）来添加支持。

随着触屏界面在移动设备上的流行，触摸事件被添加到了 Web 浏览器中。它们有点类似鼠标事件，因为支持相对于客户端、页面和屏幕的 *x* 和 *y* 坐标。但是，它们还支持一些不同的信息，尤其是因为大多数设备支持同时多点触摸（例如每个手指都触摸屏幕），触摸事件包含了不同触摸点各自的信息。

浏览器还定义了新的事件类型，表 12-2 做了概括。

表12-2：浏览器触摸事件

事　件	描　述
`touchstart`	当检测到触摸的时候触发（比如当一个手指第一次触摸屏幕）
`touchmove`	当触摸位置变化的时候触发（比如当一个手指在屏幕中移动）
`touchend`	当触摸结束的时候触发（比如手指从屏幕上移开）
`touchcancel`	当触摸移到屏幕的触摸感应区外面，或触摸被其他特殊实现行为中断（比如太多触摸点）时触发

> 完整的浏览器触摸事件规范可以在 W3C 推荐页面（http://www.w3.org/TR/2013/REC-touch-events-20131010/）找到。
>
> 注意，在有些浏览器中对触摸事件的支持还在开发中，你可能会遇到特定浏览器的问题。比如桌面版的 Internet Explorer 在支持触摸的 PC 上支持触摸事件，但它们需要不同的 DOM 事件类型和特定浏览器的 CSS 属性（`-ms-` 前缀）来让它正常工作。具体细节请查询你目标浏览器的开发者文档。

为了给 Futurgo 应用添加触摸支持，我们需要实现前面所提到的事件的处理函数。我们需要把它们添加到 Vizi 查看器所使用的模型控制器中，来支持模型的旋转和缩放。我们还想实现 Futurgo 应用自身的触摸事件支持，来处理用户触摸车身的一部分。

1. 在查看器中实现基于触摸的模型旋转

桌面 Futurgo 应用的一个良好特性是能使用鼠标旋转模型，并使用鼠标滚轮和触摸板来缩放。因为这些输入源在移动设备上都没有，所以我们将使用触摸来替代。

回忆一下第 10 章 Futurgo 应用使用 `Vizi.Viewer` 对象以及它的内建模型控制器，来支持用鼠标操作模型。我们将修改模型控制器来使用触摸输入。这个类的源码可以在 Vizi 源码文

件的 src/controllers/orbitControls.js 中找到。

首先，我们添加 `touchstart` 的事件监听函数，它将调用 `onTouchStart()` 方法。

```
this.domElement.addEventListener( 'touchstart', onTouchStart,
  false );
```

其他事件监听函数也随后都添加到 `onTouchStart()` 中了。（注意变量 `scope` 是 JavaScript 的闭包范围变量，保存的是 orbit 控制对象的 `this` 值。）

```
scope.domElement.addEventListener( 'touchmove', onTouchMove,
  false );
scope.domElement.addEventListener( 'touchend', onTouchEnd,
  false );
```

现在我们可以处理触摸事件了。例 12-1 展示了 `onTouchStart()` 的代码。我们基本上就是伪造了一个鼠标按下（mouse down）事件，并调用对应的事件处理函数 `onMouseDown()`，`onMouseDown()` 是用户鼠标事件的处理。每个触摸输入源的细节存储在事件的 `touches` 列表中，它是一个 `Touch` 对象的数组。我们这里假定只有一个触摸，忽略列表中除了第一个对象以外的其他对象。对象的值被复制到了我们伪造的鼠标事件中，并传给 `onMouseDown()`，然后这个事件被当成一个普通鼠标按下事件来处理。

用 Mr. Spock 的话来说："原始但有效。"

例 12-1：通过合成一个鼠标按下事件来处理触摸开始

```
 // 合成一个鼠标左键事件
var mouseEvent = {
    'type': 'mousedown',
    'view': event.view,
    'bubbles': event.bubbles,
    'cancelable': event.cancelable,
    'detail': event.detail,
    'screenX': event.touches[0].screenX,
    'screenY': event.touches[0].screenY,
    'clientX': event.touches[0].clientX,
    'clientY': event.touches[0].clientY,
    'pageX': event.touches[0].pageX,
    'pageY': event.touches[0].pageY,
    'button': 0,
    'preventDefault' : function() {}
    };

onMouseDown(mouseEvent);
```

我们用类似的简单技巧实现了 `touchmove` 和 `touchend`，不同的是我们使用了 `event.changedTouches` 数组。`changedTouches` 包含任意触摸输入源移动时的新值。我们同样全部假定是一个触摸操作。这是没问题的，我们对于多点触摸有其他方案。具体细节可以查看 `onTouchMove()` 和 `onTouchEnd()` 方法的代码。

2. 实现基于多点触摸的缩放

大部分设备支持多个触摸输入，或者说支持*多点触摸*操作。一个常用的多点触摸交互方式是使用两根手指来"捏合"屏幕，即将两根手指移近和移远来缩小或放大视图。我们将在

Vizi 模型控制器上进行实现。

开发多点触摸比开发单点触摸更复杂一些，因为我们需要分别跟踪多个触摸输入的移动。触摸对象在事件 `touches` 或 `changedTouches` 列表中包含一个 `identifier` 属性，它是触摸唯一的 id，在它的周期（从 `touchstart` 到 `touchmove`，直到 `touchend` 或 `touchcancel`）里保持不变。

让我们来看一下代码。在 `onTouchStart()` 的开始部分，我们检测是否有一个以上触摸。如果是的话，我们将它当成捏合（pinch-to-zoom）手势，而不是模型旋转。我们使用 `touches` 数组的前两项来计算触摸间的距离，保存它们到 `touchDistance` 属性中。我们将在后面使用它来确定是向内挤压（pinch inward）还是向外挤压（pinch outward）。

```
if ( event.touches.length > 1 ) {
        scope.touchDistance = calcDistance(event.touches[0],
          event.touches[1]);
        scope.touchId0 = event.touches[0].identifier;
        scope.touchId1 = event.touches[1].identifier;
    }
```

我们还分别在属性 `touchId0` 和 `touchId1` 里保存了两个触摸对象的字符串标识符。我们这么做是因为当接下来接收到 `touchmove` 事件时，我们必须确定哪个触摸对象移动了，因为并不能保证触摸对象在新的 `changedTouches` 事件列表中保存的顺序和最初的 `touchstart` 事件是一样的。唯一能区分每个触摸对象的信息就是它的 `identifier` 属性，所以我们保存下来供后续使用。

现在可以处理 `touchmove` 了，请看例 12-2。在 `onTouchMove()` 方法中，我们首先确定是否有多点触摸事件。如果有，我们在 `changedTouches` 中查找之前保存的两个标识符：`touchId0` 和 `touchId1`。有这些标识符的触摸对象是我们所关心的。一旦获得了它们，我们就可以使用辅助函数 `calcDistance()` 计算新的距离。我们将它和之前的距离相减：如果结果是正的，说明我们两根手指的距离远了，于是我们拉近相机来让模型看起来更大；如果结果是负的，说明我们两根手指的距离更近了，于是缩小模型。

例 12-2：使用多点触摸处理一个捏合操作

```
if ( event.changedTouches.length > 1 ) {
    var touch0 = null;
    var touch1 = null;
    for (var i = 0; i < event.changedTouches.length; i++) {
        if (event.changedTouches[i].identifier ==
          scope.touchId0)
            touch0 = event.changedTouches[i];
        else if (event.changedTouches[i].identifier ==
          scope.touchId1)
            touch1 = event.changedTouches[i];

    }
    if (touch0 && touch1) {
         var touchDistance = calcDistance(touch0, touch1);
         var deltaDistance = touchDistance -
           scope.touchDistance;
         if (deltaDistance > 0) {
```

```
                scope.zoomIn();
            }
            else if (deltaDistance < 0) {
                scope.zoomOut();
            }
            scope.touchDistance = touchDistance;
        }
    }
```

让我们看一下距离是如何计算的。例 12-3 展示了 calcDistance() 的全部代码。计算很简单，使用经典的毕达哥拉斯（Pythagorean）距离公式。

例 12-3：计算捏合的距离

```
function calcDistance( touch0, touch1 ) {
   var dx = touch1.clientX - touch0.clientX;
   var dy = touch1.clientY - touch0.clientY;
   return Math.sqrt(dx * dx + dy * dy);
}
```

3. 关闭页面的用户缩放

为了正确实现我们的触摸，还有一个很小但非常重要的细节。默认情况下，移动 Web 浏览器允许用户使用触摸来缩放页面内容。但是，这会干扰我们使用捏合来缩放 3D 内容的能力。好消息是有办法在标签中关闭这个功能。通过包含以下 HTML5 meta 标签，我们可以防止用户缩放页面（请看 Chapter 12/futurgo.html）：

```
<meta name="viewport"
  content="width=device-width, initial-scale=1.0, user-scalable=no">
```

4. 为Futurgo模型添加Vizi.Picker触摸事件

桌面版的 Futurgo 有一个非常好的功能：对于车模型不同部分的信息标注。移动鼠标悬停到车上一部分（挡风玻璃、车身、轮胎）的时候，会弹出一个关于那个部分更多信息的 DIV。但是，移动设备没有鼠标，所以基于鼠标悬停是不能工作的。作为替代，我们可以在模型的不同部分被触摸时弹出标注。Vizi.Picker 包含了对触摸事件的支持。请看文件 Chapter 12/futurgo.js 的第 44 行，我们添加了基于触摸触发标注的代码。注意例 12-4 中粗体标出的行。

例 12-4：为 Futurgo 模型添加 Vizi.Picker 触摸事件

```
// 为所有的车窗添加进入时淡出的行为
var that = this;
scene.map(/windows_front|windows_rear/, function(o) {
    var fader = new Vizi.FadeBehavior({duration:2, opacity:.8});
    o.addComponent(fader);
    setTimeout(function() {
        fader.start();
    }, 2000);

    var picker = new Vizi.Picker;
    picker.addEventListener("mouseover", function(event) {
        that.onMouseOver("glass", event); });
    picker.addEventListener("mouseout", function(event) {
        that.onMouseOut("glass", event); });
```

```
        picker.addEventListener("touchstart", function(event) {
            that.onTouchStart("glass", event); });
        picker.addEventListener("touchend", function(event) {
            that.onTouchEnd("glass", event); });
        o.addComponent(picker);
    });
```

触摸事件的处理函数很简单，我们再次使用了发送到一个现有鼠标处理函数这一简单的技巧。

```
    Futurgo.prototype.onTouchEnd = function(what, event) {
        console.log("touch end", what, event);
        this.onMouseOver(what, event);
    }
```

谢天谢地，`onMouseOver()` 中不需要一个真实的 DOM 鼠标事件（`MouseEvent`），不然这段代码会失效。我们这里做了简单的处理，但不要在你的产品代码中做类似的事情，不然你会在后面不经意间发现 bug。

12.2.2　在桌面版Chrome上调试移动功能

一旦我们学会了如何处理触摸事件，就很容易在 Vizi 核心和 Futurgo 应用中添加对它们的支持。甚至是添加多点触摸来支持捏放缩放，虽然有些细节问题，但并不高深。尽管上面的事情看起来不算难，但我们是人类，时常会犯错，所以我们需要能调试和测试我们添加的新功能。

表 12-1 中列出的每个移动 HTML5 平台都提供了在设备上连接调试器调试应用的不同方法。有些系统做得不错，而有些从我的体验上看，用起来非常痛苦。尽管如此，你有时候仍需直面那些操作流程。我们不会讨论每个不同的工具，请查询你目标平台的文档来获得更多信息。

与此同时，如果在将应用迁移到设备前，能使用应用的桌面版本来进行一些调试就好了。幸运的是，桌面版 Chrome 的调试工具提供了模拟某些移动功能的能力，比如触摸事件。当触摸事件模拟开启的时候，你可以使用鼠标来触发触摸事件，以下是快速的操作步骤。

(1) 在 Chrome 浏览器中打开你的应用。
(2) 打开 Chrome 调试工具。
(3) 点击右下角的 Settings（设置）图标（呈齿轮状），你将会看到一个用户界面面板在调试区域中弹出。如图 12-2 所示。相关的输入区域被圈了起来。
(4) 选择设置区域的（最左边一栏）的 Overrides（覆盖）标签。
(5) 勾选 Overrides 标签栏的 Enable 复选框。
(6) 向下滚动，直到你看到右边详情区域的“Emulate touch events”。勾选那个复选框。
(7) 现在你可以点击右上角的关闭来关闭面板。当然，你需要保持调试工具开启。

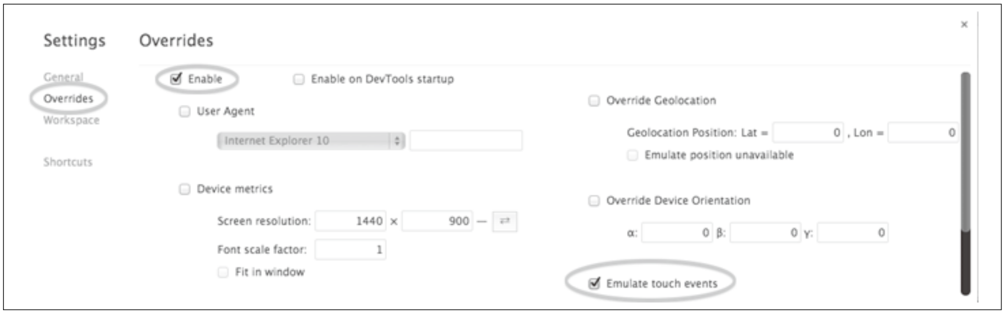


图 12-2：在桌面版 Chrome 中开启触摸事件模拟

> 注意，Chrome 触摸事件模拟只有在调试工具打开的时候才有效。当关闭调试工具后，你将失去触摸事件覆盖。

现在，浏览器触摸事件模拟已经在 Chrome 中开启了。鼠标事件将转换成触摸事件并发往你的应用。请看图 12-3。注意窗口右上角（在图中圈起来了）有红色文本的黑色矩形区域，它告诉我们事件覆盖已经开启 。现在使用鼠标点击 Futurgo，我们可以看到当 `touchstart` 和 `touchend` 事件触发时写到控制台的消息，我们将它们在控制台窗口中圈了起来。这个简单的能力是在应用迁移到移动设备前，调试触摸代码的好方法。不过，只支持单次触摸模拟。

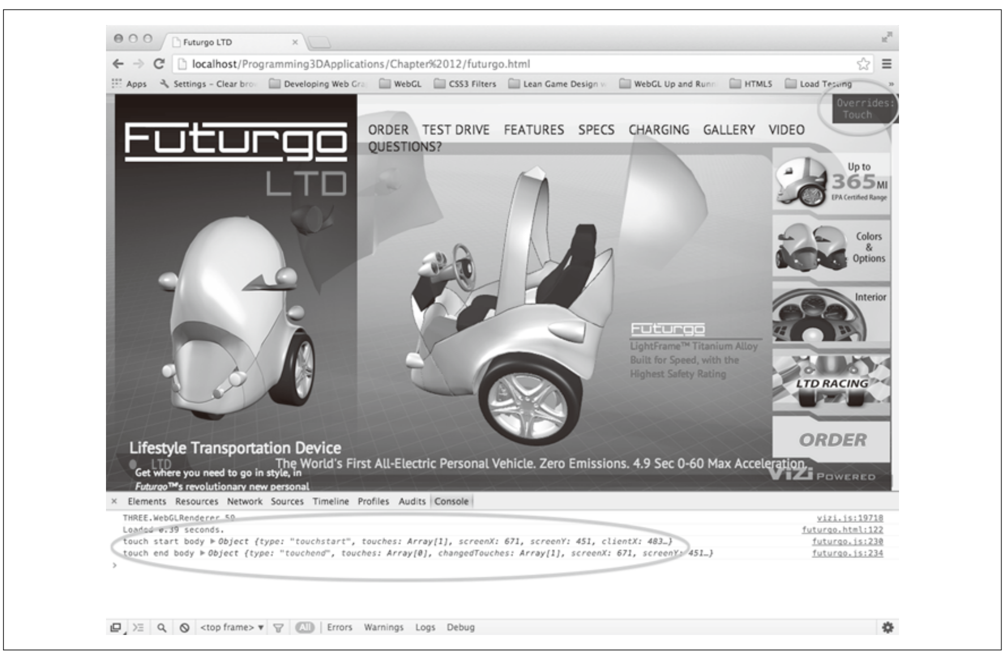


图 12-3：在桌面版 Chrome 中调试 Futurgo 的触摸事件

12.3 创建Web应用

有时候，你想将你的作品打包为一个完成的应用部署到设备上。或者你想使用应用内购买，或其他为应用而准备但在浏览器中不支持的平台功能。或者你只是想在用户的设备上安装一个图标，使得他能直接打开你的应用。大部分新的移动设备平台支持用 JavaScript 和 HTML5 开发，然后将结果打包为一个应用成品或 Web 应用。

12.3.1 Web应用开发和测试工具

使用 HTML5 创建 Web 应用的开发工具因平台而异，每个平台都有自己的测试、调试和打包发布应用的方法。

Amazon 为在 Kindle 设备上的 Amazon Fire OS 提供了一个 Web App Tester 测试工具。Fire OS 是 Amazon 为 Kindle Fire 设备开发的操作系统，其基于 Android。Web App Tester 是一个 Kindle Fire 应用，在 Amazon 商店中有提供。详细信息可以访问 https://developer.amazon.com/sdk/webapps/tester.html。Web App Tester 如图 12-4 所示。



图 12-4：Amazon Web App Tester

这个工具的使用非常简单：在你的页面中输入一个 URL，它就会以全屏的方式来加载这个页面。当你首次输入这个 URL 之后，它就会被保存在测试商店中以便再次载入。

> 之前提到过，创建 Web 应用的开发者工具因平台而异。甚至对于不同厂商版本的 Android 也是如此：尽管 Kindle Fire OS 是基于 Android 的，但 Amazon 添加了大量功能，有一系列自定义的工具来开发、测试和打包。对于其他基于 Android 的系统，请查询厂商的文档或查看 Android Web 应用开发者页面（http://developer.android.com/guide/webapps/index.html）。

12.3.2 打包成Web应用来发布

一旦你调试和测试完你的应用，就可以发布它了。同样，每个平台的发布流程也都不同。Amazon 提供了移动应用分发入口，允许注册的 Amazon 开发者创建 Kindle Fire 和 Android 应用来发布。通过这个入口发布你的应用需要经过几个步骤。第一步是为应用创建一个清单文件（manifest file），这是一个包含关于应用内容及功能的信息的文件。下面是一个示例，是一个非常简单的 Amazon Web 应用的清单文件。唯一需要的字段是 `verification_key`，它是在 Amazon 发布流程中生成的值。其他关于应用的元数据，比如图标和描述，在提交应用的时候在线提交，而不在 manifest 文件中。

Amazon 清单文件的完整信息可以在 https://developer.amazon.com/sdk/webapps/manifest.html 中找到。

```
{
    "verification_key":
      "insert your verification key from the App File(s) tab",
    "launch_path": "index.html",
    "permissions": [
        "iap",
        "geolocation",
        "auth"
        ],

        "type": "web",
        "version": "0.1a",
        "last_update": "2013-04-08 13:30:00-0800",
        "created_by": "webappdev"
    }
```

作为对比，Firefox OS 的 Firefox 市场有一个不同的应用分发流程，以及不同语法的清单文件。下面是一个简单的例子：

```
{
  "name": "Your_Application_Name_Here",
  "description": "Your Application Description Here",
  "version": 1,
  "installs_allowed_from": ["*"],
  "default_locale": "en",
  "launch_path": "/index.html",
  "fullscreen": "true",
```

```
    "orientation": ["landscape"],
    "icons": {
      "128": "/images/icon-128.png"
    },
    "developer": {
      "name": "Your Name Here",
      "url": "http://your.company.url.com"
    }
  }
```

在 Firefox 的清单文件中，我们列出了包中的文件、一个应用图标、一个应用名称和描述，以及一些开发者信息。在 https://developer.mozilla.org/en-US/Apps/Developing/Packaged_apps 上查找更多关于 Firefox OS 应用创建的信息。

12.4 开发原生/HTML5混合应用

HTML5 和移动平台 API 是对立的两方。我们可以想象不远的未来，任意应用都可以先在 HTML5 中开发，然后简单地使用厂商的打包技术发布到不同的移动平台上。然而，这个未来还没到来。平台间还有不同，尤其是 3D 还有差距。就像前面我们说过的，WebGL 已经在几乎所有移动浏览器中得到了支持，但并未完全普及。

因为这样或那样的原因，你需要考虑使用某种在目标设备上开启 WebGL 的技术，它通过提供原生库或*适配器*来让 JavaScript 访问 WebGL API。使用一种适配器技术，你可以在打包的应用中将 JavaScript/HTML 与原生代码相结合，变成一个混合应用，鱼和熊掌兼得。

开发者或许会因为如下原因之一而转向混合方式。

- *缺乏浏览器支持*

  iOS 是实现 WebGL 的阻碍，而且是一个大的阻碍。对于类似 iOS 及其他不支持 WebGL 的移动平台（比如早期版本的 Android）来说，混合方案提供了一种方式，允许开发者使用 JavaScript 基于 WebGL API 开发 3D 应用，并发布到目标平台。

- *性能*

  适配库通常会提供比基于浏览器 WebGL 的同等应用略好的性能。这有两个原因。首先，它们可以提供一个优化并调优的 JavaScript 虚拟机。其次，它们可以避免浏览器安全模型所添加的 WebGL 安全层——这对于基于 Web 的应用是必需的，但对于原生应用则不是。

- *发布为一个应用*

  如果目的是将应用成品发布为移动应用，而不是基于浏览器的站点，混合方案值得一试。有些混合方案甚至允许 JavaScript 使用在纯基于浏览器的应用中没有的功能，比如应用内购买、原生广告 SDK、消息推送。

近几年来，出现了几种支持混合方案的适配技术。尽管它们中的大部分提供了硬件加速的 Canvas 及其他特殊功能（其中最著名的是 Adobe 的 PhoneGap，网址是 http://phonegap.com/），但只有少数混合框架包括了对 WebGL 的支持。最值得注意的两个是 CocoonJS 和 Ejecta。尽管这两个工具试图解决相同的问题，但它们的的方法却很不同，以下是一个简

要的比较。

- CocoonJS（http://www.ludei.com/tech/cocoonjs）

  CocoonJS 可以运行在 Android 和 iOS 上。它提供了一个简单的应用容器来运行 HTML5 和 JavaScript 代码，隐藏了底层系统的细节。它提供了对 Canvas、WebGL、Web Audio、Web Sockets 等的实现。CocoonJS 还有一个云端的项目编译系统，所以你要做的仅仅是登录和构建你的项目。开发者不需要了解如何使用原生平台工具，比如 iOS 的 Xcode，来创建原生应用。CocoonJS 是闭源项目，由它位于旧金山的开发商 Ludei 严格控制。

- Ejecta（http://impactjs.com/ejecta）

  Ejecta 是一个开源库，提供了许多和 CocoonJS 一样的功能，但只支持 iOS。Ejecta 诞生自 ImpactJS，一个 HTML5 游戏引擎项目。Ejecta 更加 DIY，需要开发者对 Xcode 和原生平台 API 有相当多的了解。

尽管 Ejecta 是一个开源项目，但它依赖 iOS 特性及 Xcode，因而不适合本书。我们将使用 CocoonJS 来开发一个混合应用。

12.4.1　CocoonJS：一种为移动设备构建HTML游戏及应用的技术

CocoonJS 是一种让 HTML5 混合应用运行在移动设备上的适配器技术。它作为一个 HTML5 原生封装来工作：应用或游戏是以一个原生应用来执行的，在其中执行 JavaScript 和 HTML。CocoonJS 可以运行在 iOS 和 Android 上，在这些平台和不同设备上提供了一致的环境。

CocoonJS 允许开发者提供一个 HTML 文件及关联的 JavaScript 代码，使用标准 2D 和 WebGL 来全屏渲染 2D 或 3D Canvas，还支持 Web Audio、图片加载、Ajax 开发的 `XMLHttpRequest`、WebSockets。CocoonJS 实现了原生、硬件加速的 API，并提供了一个自定义的 JavaScript 虚拟机（由 Ludei 调优）来提供更好的性能。图 12-5 展示了 Futurgo 作为全屏原生应用运行在 Apple iPad 4 上的截图。



图 12-5：Futurgo 作为一个原生 iPad 应用在运行，基于 Ludei CocoonJS 开发

为了让开发和测试更简单，CocoonJS 提供了一个 Launcher 应用，允许开发者载入一个 URL 来预览效果，或者向 Launcher 提供一个包含所有内容的 ZIP 压缩包来在设备上运行。CocoonJS Launcher，如图 12-6 所示，可以从苹果和 Android 应用商店中下载。为了测试你的应用，可以点击 Your App 按钮，然后在文本窗口中输入测试文件的 URL，或者打开使用 iTunes 或 Android SDK 工具上传到 Launcher 中的 ZIP 文件。详细信息请参考 CocoonJS 文档。

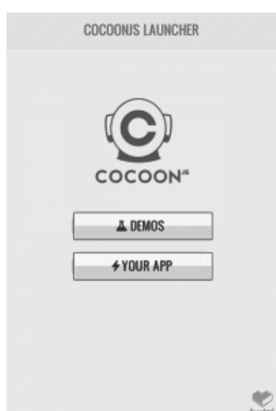


图 12-6：CocoonJS Launcher 的主屏幕

当你使用 Launcher 预览和测试应用后，可以使用 Ludei 基于云的服务来将它编译为原生应用：上传应用文件，几分钟后你就可以下载一个能在 iOS、Amazon、GooglePlay 及其他应用商店上发布的最终包。

12.4.2 使用CocoonJS组装应用

尽管 CocoonJS 的开发者声称可以用它来构建任意类型的原生 /HTML5 应用，但他们目前重点关注的是开发高性能游戏。为此，让我们开发一个非常简单的游戏来展示如何构建一个应用。这更像是一个游戏演示而不是完整游戏，主要是用于展现流程。在我们学习应用的 CocoonJS 细节前，先看看游戏的桌面版本。然后我们会使用 CocoonJS 来改造它。

在你的浏览器中打开文件 Chapter 12/omegacity/omegacity.html，你会看到如图 12-7 所示的开始画面。这个模型看起来眼熟，它是使用第 8 章的示例程序所加载的 glTF“虚拟城市”场景（Chapter 8/pipelinethreejsgltfscene.html）。



图 12-7：*Omega City* 游戏开始画面——2D 素材由 GameSalad（http://www.gamesalad.com/）设计，虚拟城市场景来自 3DRT（http://3drt.com/store/free-downloads/33-sci-fi-skyscrapers-collection.html），声音来自 FreeSound（http://www.freesound.org/）；所有版权保留

> 这个了不起的模型是从 3DRT（http://www.3drt.com/）上购买的。这里展示的 demo 是我们同奥斯汀的 GameSalad（http://www.gamesalad.com/）公司合作开发的。该公司发布过一个易于使用的 2D 游戏创建工具，可以用于 HTML 及移动游戏。记住，像本书发布的其他模型一样，这个模型是有版权保护的，因此你不能在你自己的应用中使用，或者用于学习本书外的其他用途，除非你购买自己的模型。

欢迎来到 Omega 市，银河的边防哨所。你和你的小分队是人类对付外星人进攻的最后希

望。为了拯救这个城市……你或许不得不毁灭它！

点击闪烁的 START 标签来进入游戏，它会把你带到图 12-8 所示的主界面。飞船是自动驾驶的，你只能发射武器。按下键盘的向上箭头来发射激光，你会看到蓝色的激光会聚到视图中央。按下空格键来发射导弹。在通电声音后，导弹会从飞船中心发射，当它击中目标的时候，会在一道绿色闪光中爆炸。这一简单的例子只是设计用来展示如何开发类似的游戏应用，让我们体验一下使用 CocoonJS 来进行混合 iOS 开发。

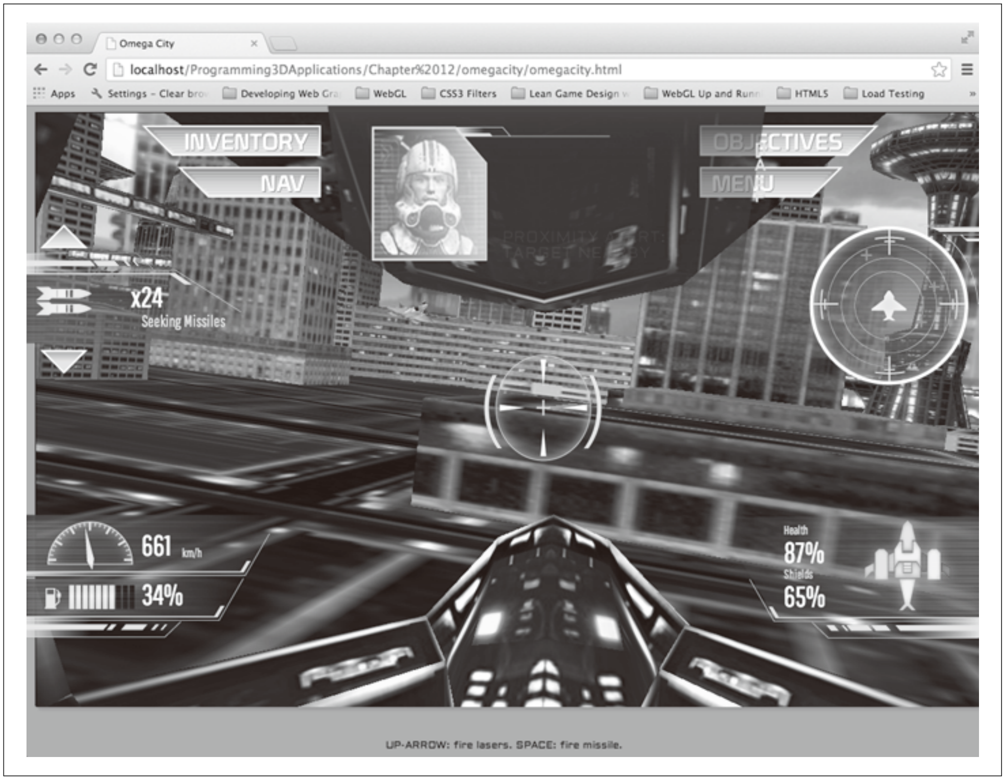


图 12-8：运行在桌面的 *Omega City* 游戏演示

1. 创建主视图和覆盖视图

你或许已注意到，Futurgo 运行在 Kindle Fire HDX 上的纯 HTML5 版本（图 12-1），与使用 CocoonJS 运行在 iOS 上的原生版本（图 12-5）有微妙的不同。Kindle Fire 版本看起来和桌面版本一样，在 3D 模型后面有紫色的背景渐变，还包含了网格线框，而 CocoonJS iOS 版本则是黑色背景。这是因为 CocoonJS 不是一个完整的 HTML5 浏览器和合成引擎，而是一个原生实现的 Canvas 元素，用于让 JavaScript 开发者可以开发原生 2D 和 3D 图形。CocoonJS 可以解析 HTML 标签，但它忽略大部分标签和样式信息。

CocoonJS 解析 HTML 标签来查找主 canvas，再加上其他关联的 JavaScript 文件，但就这些了，你不应该期望 CSS 样式都会生效。Futurgo 的背景图片是在 CSS 中通过 `container` DIV 元素指定的，而 CocoonJS 在解析 HTML 文件的时候忽略了它。如果我们希望这样的背景效

果在 CocoonJS 中生效，必须自己在 canvas 上绘制它。对于这个例子而言，我们可以在背景添加一个 Three.js 对象，或者一个 Vizi skybox（请查看第 11 章）。对于我们这个简单快速的例子来说不值得这么做，但如果需要的话，你要在你自己的应用中处理这个问题。

尽管 CocoonJS 欠缺在背景元素样式方面的支持，但 Ludei 意识到了这样一个实际问题：许多 Web 开发者更想使用 HTML 来布局和编写游戏的用户界面。因此它提供了层叠第二个 HTML 文件的方法，将第二个 HTML 文件渲染在一个 WebView 窗口中，在主 canvas 之上。这个方法生效的关键是，将 canvas 和覆盖的 HTML 元素分成两个不同的文件或视图。图 12-9 并排显示了两个视图的内容。



图 12-9：左图为 CocoonJS 版本 *Omega City* 的 canvas 视图；右图为对应的覆盖视图

为了用 CocoonJS 来改造 *Omega City*，我们首先分离原始文件 omegacity.html 为两个不同的文件。新的文件是 index.html 和 wv.html。index.html 包含了主 canvas 的代码。wv.html 包含覆盖视图的代码。当文件分离后，我们添加 Ludei 提供的 CocoonJS 特有的辅助代码。这些文件将使用一个 WebView 控件来管理添加覆盖视图，并提供让两个视图可以通信的设施——我们在后面会详细介绍。CocoonJS JavaScript 库设计为还可以在桌面浏览器中工作，所以我们可以在桌面 Chrome 中预览结果，然后再到 CocoonJS 的 Launcher 应用中测试。

主 canvas 视图的代码可以在文件 Chapter 12/omegacity-iOS/index.html 中找到，参见例 12-5。首先，载入一些 CocoonJS 特有的文件。然后在页面加载后，构建将渲染到主 WebGL canvas 上的游戏对象。游戏对象的源码可以在文件 Chapter 12/omegacity-iOS/omegacity.js 中找到。之后创建一个对象来管理覆盖视图或平视显示器（HUD，Heads-Up Display），以及创建另外一个对象来管理游戏中的声音。（注意 HUD 类的 proxy 前缀，我们马上会介绍它。）

例 12-5：CocoonJS 应用主视图代码

```
<script src="./libs/cocoon_cocoonjsextensions/CocoonJS.js">
</script>
<script src="./libs/cocoon_cocoonjsextensions/CocoonJS_App.js">
</script>
<script
src="./libs/cocoon_cocoonjsextensions/CocoonJS_App_ForCocoonJS.js">
</script>
```

```
<script src="./libs/vizi/vizi.js"></script>
<script src="omegacity.js"></script>
<script src="omegacityProxyHUD.js"></script>
<script src="omegacitySound.js"></script>
<script>

   var game = null;
   var hud = null;
   var sound = null;
   var gameLoadComplete = false;
   var wvLoadComplete = false;

var handleLoad = function() {

    var container = document.getElementById("container");

    game = new OmegaCity({ container : container,
            loadCallback : onLoadComplete,
            loadProgressCallback : onLoadProgress,
         });

    hud = new ProxyHUD({game : game});

    sound = new OmegaCitySound({game : game});
```

创建游戏对象后，我们接着调用 CocoonJS 应用的 `loadInTheWebView()` 方法来加载 wv.html 文件中的覆盖视图：

```
    setTimeout(function() {

        CocoonJS.App.onLoadInTheWebViewSucceed.addEventListener(
                function(url) {
                    CocoonJS.App.showTheWebView();
                    Vizi.System.log("load web view succeeded.");
                    wvLoadComplete = true;

                    if (gameLoadComplete) {
                        gameReady();
                    }
                }
                );
        CocoonJS.App.onLoadInTheWebViewFailed.addEventListener(
                function(url) {
                    Vizi.System.log("load web view failed.", url);
                }
                );
        CocoonJS.App.loadInTheWebView("wv.html");
    }, 10);

    sound.enterState("load");
    game.load();

}
```

覆盖视图包含了所有的 HTML 及 JavaScript 代码（例 12-6）。请打开文件 Chapter 12/

omegacity-iOS/wv.html 查看。在 HUD 对象的标签后，我们包含了 CocoonJS 特有的文件来帮助管理视图。然后我们创建一些自己的对象，但它们只用在用户界面中。（这里也有代理对象，这次是游戏和音乐对象。稍后进行介绍。）

例 12-6：CocoonJS 应用的覆盖视图代码

```
    <!--  加载信息  -->
    <div id="loadStatus" style="display:none">
    Loading...
    </div>
    <!--  Click-to-start screen  -->
    <div id="startScreen" style="display:none">
    <!--  Logo -->
    <div id="logowtext"></div>
    <div id="startScreenText">
    Welcome to Omega City, the frontier outpost of the galaxy.
    You and your squad are humanity's
    last best hope as aliens attack.<br></br>
    To save the city... you may have to destroy it!
    </div>

... <!-- 其他标签 -->
    <script src="./libs/cocoon_cocoonjsextensions/CocoonJS.js">
    </script>
    <script src="./libs/cocoon_cocoonjsextensions/CocoonJS_App.js">
    </script>
    <script
    src="./libs/cocoon_cocoonjsextensions/CocoonJS_App_ForWebView.js">
    </script>
    <script src="omegacityGameProxy.js"></script>
    <script src="omegacityProxySound.js"></script>
    <script src="omegacityHUD.js"></script>
    <script>

       var hud = null;
       var game = null;
       var sound = null;

    var onload = function() {

        hud = new OmegaCityHUD();
        sound = new ProxySound();
        game = new OmegaCityGameProxy();
    }
```

我们需要再做一件事来将这两个视图连接起来：保证我们的视线可以穿透覆盖视图。所以我们修改覆盖视图的 CSS，设置所有 body 元素的背景颜色为透明。请查看文件 Chapter 12/omegacity-iOS/css/omegacity.css。下面是 CSS：

```
     body {
        background-color:rgba(0, 0, 0, 0);
        color:#11F4F7;
        padding:0;
        margin-left:0;
        margin-right:0;
```

```
    overflow:hidden;
}
```

2. 管理canvas和覆盖视图间的通信

CocoonJS 提供的覆盖 Web 视图是通过一个覆盖在主 CocoonJS canvas 视图上的 WebView 控件实现的。这个架构有个主要的隐含之意：canvas 视图的 JavaScript 虚拟机完全和 WebView 中脚本的虚拟机分离。换句话说，这两个脚本引擎执行在不同的环境中，很可能使用两个完全不同的 JavaScript 虚拟机！主视图使用的是 CocoonJS 的虚拟机，而在上面的 WebView 控件使用的是来自平台的原生脚本引擎。如果主视图中的代码试图调用覆盖视图的函数，它会失败；反之亦然。但是，CocoonJS 提供了让两个视图通过发送消息来进行通信的方法。并且幸运的是，它不需要我们理解细节。

CocoonJS 提供了另一个应用方法 `forwardAsync()`，它允许我们在两个环境间传递字符串。字符串会使用 JavaScript `eval()` 执行。所以调用另一个环境的函数，只需创建一个那样的字符串，当求值（evaluate）的时候再调用相关函数。

为了让这样的代码更可读，我们将包装每个 `forwardAsync()` 调用为代理（proxy）对象的直接方法调用：调用代理对象的方法时，内部会调用 `forwardAsync()`，它然后会向另一个（远程）环境发消息。当消息求值后，在另一个环境中的函数被调用了，它最终可以调用远程对象的方法。

为了说明这一点，让我们看一下点击 START 标签开始游戏的代码。这个代码在文件 Chapter 12/omegacity-iOS/omegacityProxyHUD 中，展示了一个来自 `OmegaCityGameProxy` 类的方法，它从覆盖视图向主视图发送一个消息：

```
OmegaCityGameProxy.prototype.play = function() {
    CocoonJS.App.forwardAsync("playGame();");

}
```

主视图的代码接收 `playGame()` 消息，告诉声音引擎播放主游戏声音，然后告诉真正的游戏对象来开始游戏。

```
    function playGame() {
       sound.enterState("play");
       game.play();
    }
```

在另一个方向上，有游戏内部发生的事件来更新显示，比如当一个火箭发射后减少火箭计数。还有当外星船接近的时候，设置一个接近警告，用新的红色闪烁文字更新顶部的消息区域。我们通过使用 HUD 的一个代理对象，从另一个方向发送消息（从主视图到覆盖视图），来实现这些 HUD 的方法。

```
ProxyHUD.prototype.enterState = function(state, data) {

    CocoonJS.App.forwardAsync("hudEnterState('" + state + "','" +
      data + "');");

}
```

覆盖层的代码然后通过调用真实 HUD 对象的 `enterState()` 方法来处理 `hudEnterState()` 消息：

```
function hudEnterState(state, data) {
   console.log("HUD state: " + state + " " + data);
   hud.enterState(state, data);
}
```

> 刚才展示的设计模式或许看起来奇怪，但事实上它在支持进程间通信（InterProcess Communication，IPC）的系统中相当常见。它们使用诸如远程过程调用（Remote Procedure Call，RPC）的技术，其中有两个独立的计算机进程，通过封装函数调用的消息来相互通信。
>
> CocoonJS 的双视图架构，使得我们如果想在混合应用上构建一个基于 HTML5 的覆盖层，就必须使用 RPC。编写双向通信的代理代码有点乏味，如果有自动化工具会容易些。在我和 Ludei 开发者的讨论中，他们暗示了相关工具正在开发中。

12.4.3 WebGL混合开发：问题

在这一节中，我们研究了使用混合方法开发一个基于 HTML5 的移动 3D 应用：一个原生应用，使用 WebView 来渲染 HTML，加上原生的库来模拟 WebGL API。对于类似 iOS 这样的环境（移动版 Safari 和移动版 Chrome 浏览器不支持 WebGL），混合方法是值得考虑的。

我们研究了用 Ludei 的 CocoonJS 作为一种可能的混合方案。CocoonJS 允许我们轻松组装应用，不需要我们学习像 iOS 的 Cocoa 那样的原生 API。不过我们还需要做一个额外的工作来开启 HTML5 覆盖视图。因为 CocoonJS 并不是一个完整的 Web 浏览器，它只是一个 canvas 渲染器，所以我们需要分离所有 HTML5 UI 元素到另一个 WebView 控件上，使用特殊的 JavaScript API 在那个视图及 canvas 间进行通信。尽管这个方案有局限性，但它对大部分使用场景已经足够了。然而 CocoonJS 并不开源，它的开发公司还在积极研究对开发者的授权方式。一个开源的替代方案是 Impact Ejecta，但使用这个库需要大量的 iOS 开发知识。而且它还在开发中，缺乏打磨。

使用 3D 混合开发的问题是没有一个理想的方案。但它有可行的开发方案，最终要取决于你的需求和预算。

12.5 移动3D性能

移动平台的资源比桌面平台有限，它们通常内存更小，CPU 和 GPU 的性能更差。移动平台还会有带宽方面的限制，这取决于设备的网络类型及流量套餐。无论你开发的是一个基于浏览器的 Web 应用、打包为 Web 应用的纯 HTML5 应用，还是使用 CocoonJS 或 Ejecta 的原生 /HTML5 混合应用，你都需要在开发移动 3D 应用时特别关注性能问题。

关于性能问题的完整处理方法超出了本书的范围，我们可以快速浏览一些更突出的问题，并介绍几种技术来作为备用。以下是需要注意的性能问题，排序不分先后。

- *JavaScript 内存管理*

  JavaScript 是自动垃圾回收的语言。这意味着开发者不需要显式分配内存，而是由虚拟机来完成。它还会在不需要使用内存的时候将其释放供后续使用，这个过程被称为*垃圾回收*（garbage collection）。垃圾回收被设计为由虚拟机来决定在何时执行。这会导致应用在虚拟机需要花时间进行垃圾回收的时候，遭遇明显的延迟。有许多减少虚拟机在垃圾回收方面所耗时间的技术，列举如下。

  - 在应用启动的时候预先分配好内存。
  - 创建可重用的对象“池”，可由开发者来进行循环使用。
  - 函数在返回复杂值的时候，不是创建一个新的 JavaScript 对象，而是放在合适的对象内传递过来。
  - 避免闭包（即对象可以持有函数作用域外的其他对象）。
  - 一般来说，避免在不必要的时候使用 `new` 操作符。

  移动平台更会受到垃圾回收的影响，因为它们内存更少。

- *性能更差的 CPU 和 GPU*

  制造商让移动设备更轻和更便宜的一种方法是使用性能更差和更便宜的组件，包括 CPU 和 GPU。尽管移动平台开始变得惊人地强大，但它们还比不上桌面平台。为了更好地支持小的 CPU 和 GPU，并提供更好的用户体验及节省电池寿命，请考虑以下策略。

  - *使用低分辨率的 3D 内容*。3D 内容会对移动设备的 CPU 和 GPU 造成负担。尤其是手机，没必要发布非常高的分辨率，因为它们屏幕的分辨率并不高。为什么要浪费额外的分辨率呢？这一技术还可以通过减小下载大小，来帮助减轻低速网络的负荷。另一方面，新的平板提供了非常高的分辨率，因此你需要小心地做好平衡。
  - *注意你的算法*。一台非常快的机器可以掩盖糟糕的代码，但是，移动设备往往会将它们暴露无遗。比如，试着在 Kindle Fire HDX 上点击 Futurgo 的金属车身。有时候你会看到停断，这是代码在查找哪个对象被点击了。这是 Three.js 内选取实现的副作用。它的代码使用未优化的算法，在小设备上便暴露出来了。有一天，这个代码会在 Three.js 中被修复，或在类似 Vizi 那样的框架中以另一种更好的方式实现，但现在，需要注意类似这样的性能问题，如有必要的话，你需要解决它们来减少处理器的负担。
  - *简化着色器*。基于 GLSL 的着色器可以很复杂，复杂到编译的代码在有些性能有限的移动设备上难以支持。你需要注意在部署到这些平台上时简化着色器。

- *有限的网络资源*

  对于使用移动网络或有流量套餐限制的设备，最好试着减少数据传输的开支。3D 内容很丰富，会需要下载更多的内容。在设计你的应用时请考虑以下这些方法。

  - *预先打包资源*。如果你能打包发布 Web 应用，那就再理想不过了。内容只有在应用安装的时候下载。
  - *使用浏览器缓存*。如果可能的话，设计你的资源来利用浏览器缓存，避免不必要的下载。

- 批量资源。这个经典的 Web 性能优化技术，可以节省网络请求和服务器往返。如果发布多个位图，比如实现一个进度条，你要考虑合并这些位图到一个 CSS 图片精灵（sprite）中（即所有图片存储在一个文件中，再在 CSS 中定义偏移）。
- 使用二进制格式和数据压缩。第 8 章讨论的 glTF 文件格式的一个主要的目的是减少文件大小，从而减少下载时间。它是通过使用二进制格式实现的。这个技术可以与服务器端压缩，甚至特定领域压缩算法相结合，比如 3D 几何体压缩，来进一步减少下载次数和减轻数据网络的负担。

12.6 小结

本章探索了使用 HTML5 和 WebGL 开发移动 3D 应用的全新世界。移动平台在性能上开始逐渐比肩桌面平台，同时，HTML5 受到移动设备影响引入了新的强大功能。大部分移动平台支持 3D：CSS3 无处不在，WebGL 在除 iOS 上移动版 Safari 和移动版 Chrome 之外的平台上都被支持。

移动浏览器 WebGL 的开发流程非常简单。现有的应用通常不需要修改就能工作。但是基于鼠标的输入必须替换为触摸输入。我们研究了如何添加触摸事件到 Vizi 查看器中实现旋转和缩放功能。我们还为 Futurgo 模型添加了轻拍支持，使得点击车的不同部分会展现弹出层。为了在桌面上开发和测试触摸功能，我们设置桌面 Chrome 来模拟触摸事件。我们还可以通过不同平台厂商提供的打包和分发技术（比如 Amazon 的移动应用分发中心），使用 WebGL 代码来创建打包的 3D 应用（即平台的“Web 应用”）。

对于不支持 WebGL 的平台，我们使用类似 CocoonJS 和 Ejecta 的适配技术来创建混合应用，结合 HTML5 与原生代码。这允许我们用 JavaScript 构建，并发布高性能、兼容各个平台的原生应用，并可能可访问只在原生平台上的功能，比如应用内购买和消息推送。

最后，我们快速浏览了移动性能问题。尽管移动平台在这几年进步地非常快速，但它们依然比桌面系统性能差。我们需要留意性能问题，具体来说是内存管理、CPU 和 GPU 的使用，以及带宽，并对此进行相应的设计。

附录

资源

本附录按类别列出了 3D Web 开发的资源。我经常访问以下大部分站点来查找最新的技术资讯、库、工具、前沿演示，以及 3D 开发社区领袖们的想法。

A.1　WebGL资源

A.1.1　WebGL规范

WebGL 标准由 Khronos 组织开发和维护，该组织还掌管着 OpenGL、COLLADA，以及其他你或许听过的规范。你可以在 Khronos 网站（http://www.khronos.org/registry/webgl/specs/latest/1.0/）找到最新版本的官方 WebGL 规范。

A.1.2　WebGL邮件列表和论坛

Khronos 维护了一个公开的邮件列表，来讨论 WebGL 规范的草图。你可以按照 https://www.khronos.org/webgl/public-mailing-list/ 的指示来订阅 public_webgl@khronos.org 邮件列表。

还有一个 Google group 来讨论核心规范外更通用的 WebGL 开发话题。你可以在 http://goo.gl/CJIvC4 注册这个列表。

A.1.3　WebGL博客和演示站点

有许多出色的博客致力于探讨 WebGL 开发。以下是一些我经常访问的博客。

- Learning WebGL（http://learningwebgl.com/blog/）
  WebGL 网站的祖师爷，由 Giles Thomas 创建，现在是我在维护。这应该成为你学习底

层 WebGL 开发的基础知识及 API 用法的第一站。它还有最新 WebGL 演示及开发项目的每周摘要。

- Learning Three.js（http://learningthreejs.com/）
  Jerome Etienne 的博客，专注于 Three.js 技术及开发实践。
- TojiCode（http://blog.tojicode.com/）
  Google 工程师 Brandon Jones 的博客，特点在于有大量关于 WebGL API 的深入的技术信息，以及专业的开发话题。
- Three.js on Reddit（http://www.reddit.com/r/threejs）
  Three.js 的 Reddit，由 Theo Armour 维护，更新频繁。这个 Reddit 有许多演示、技术、新闻及文章。
- WebGL.com（http://www.webgl.com/）
  由来自纽约的 Darien Acosta 创建，它是一个发现新 WebGL 游戏、演示及应用的网站。
- WebGL Mozilla Labs Demos（https://developer.mozilla.org/en-US/demos/tag/tech:webgl/）
  Mozilla Labs 及其合作伙伴创建的演示。
- WebGL Chrome Experiments（http://www.chromeexperiments.com/webgl）
  Google 及其合作伙伴创建的演示。

A.1.4 WebGL社区站点

我主持了湾区的 WebGL 聚会（http://www.meetup.com/WebGL-Developers-Meetup/）。在洛杉矶、纽约、波士顿、伦敦及其他地方都有 WebGL 聚会。聚会是将志同道合的人联系在一起的好方式。如果你不住在旧金山，可以在 http://Meetups.com 查找你区域的 WebGL 组，或者自己组织。

此外还有一个 LinkedIn 群组（http://www.linkedin.com/groups?gid=2426944）和一个 Facebook 主页（http://www.facebook.com/groups/webgl/）。

A.2 CSS3资源

A.2.1 CSS3规范

万维网联盟（W3C）维护 CSS3 规范的核心，包括 3D 变换、过渡、动画和滤镜效果：

http://www.w3.org/TR/css3-transforms/
http://www.w3.org/TR/css3-transitions/
http://www.w3.org/TR/css3-animations/
http://www.w3.org/TR/filter-effects/

第 6 章介绍的 CSS 自定义滤镜的主要支持者是 Adobe。它在浏览器里并没有得到广泛支持——目前仅有 Chrome 支持——所以你使用它开发的时候要注意了。最新的信息可以在

http://adobe.github.io/web-platform/samples/css-customfilters/ 找到。

A.2.2 CSS3博客和演示站点

目前在 Twitter 工作的 David DeSandro，创建了理解如何使用 CSS 3D 变换的最好资源（http://24ways.org/2010/intro-to-css-3d-transforms/）。

Codrops（http://tympanus.net/codrops/）是一个 Web 设计和开发博客，有几个极好的 CSS 3D 效果演示，包括第 6 章的重头戏——3D 书展示（http://tympanus.net/codrops/2013/01/08/3d-book-showcase/）。

Dirk Weber 的 HTML5 开发网站（http://www.eleqtriq.com）有几个引人入胜的 CSS 3D 演示。

Keith Clark 挑战了 CSS 的极限，完全用 CSS 3D 创建了一个令人兴奋的第一人称射击演示（http://blog.keithclark.co.uk/creating-3d-worlds-with-html-and-css/）。

微软的 Kirupa Chinnathambi 提供了 CSS 过渡及动画的深入信息，具体请查看文章 https://www.kirupa.com/html5/all_about_css_transitions.htm 和 https://www.kirupa.com/html5/all_about_css_animations.htm。

Bradshaw Enterprises（http://css3.bradshawenterprises.com/）上有一些学习 CSS3 过渡、变换、动画及滤镜效果的有价值的文章、教学和资源。

A.3 Canvas资源

A.3.1 Canvas 2D Context规范

2D Canvas API 规范由 W3C 维护，你可以在 http://www.w3.org/TR/2dcontext2/ 找到最新的规范。

A.3.2 Canvas 2D教程

第 7 章讨论过，开发者可以使用 Three.js 或 K3D/Phoria（稍后介绍）来创建由 2D Canvas API 渲染的 3D 应用。这些库隐藏了 2D Canvas 渲染的细节，提供了高层次的 3D 组件。但是，如果你想学习 2D Canvas API 的细节，网上有大量资源。以下是我在写作本书时发现的几个非常有用的链接：

https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial
http://www.w3schools.com/html/html5_canvas.asp
http://diveintohtml5.info/canvas.html

A.4　框架、库和工具

A.4.1　3D开发库

这几年出现了若干开源 3D JavaScript 库，我将其中一些优秀的库列举如下，排名不分先后。

- Three.js（http://threejs.org/）

  目前为止开发 WebGL 应用最流行的场景图库。Three.js 已经被用来开发许多知名的旗舰 WebGL 演示。它提供了在 3D 图形中常见的一系列简单直观的对象。它性能卓越，使用了图形引擎中的许多最佳实践技术。它功能强大，有内建对象类型和便利的工具。Three.js 还有插件渲染系统，允许将 3D 内容渲染到（有些限制）2D Canvas API、SVG 及使用 3D 变换的 CSS3。Three.js 维护良好，有好几个作者在进行贡献。

- SceneJS（http://www.scenejs.org/）

  一个开源的 JavaScript 3D 引擎，提供了 WebGL 上基于 JSON 的场景图 API。SceneJS 专门用来高效渲染大量独立可选取且有关联的对象，这在工程和医学中的高细节模型查看应用中是必需的。SceneJS 还支持物理，并提供了某些比 Three.js 更高层次的构建方法，比如事件模型和 jQuery 式的场景图 API。

- GLGE（http://www.glge.org/）

  GLGE 是一个 JavaScript 库，意图方便 WebGL 的使用和缩短设置时间，使得开发者可以将精力用在创建 Web 富内容上。GLGE 对于基本操作有良好的支持，但不像 Three.js 或 SceneJS 那样功能强大。

- K3D 和 Phoria（http://www.kevs3d.co.uk/dev/phoria/）

  K3D 以及它的后续 Phoria，只使用 2D Canvas API 渲染 3D 图形。Phoria 由英国的 Kevin Roast（http://www.kevs3d.co.uk/dev/；Twitter 账号是 @kevinroast）所创建。Kevin 是一个 UI 开发者及图形爱好者。尽管 Phoria 还处在开发早期，并不像 Three.js 那样功能强大，但它令人印象非常深刻。尤其值得一提的是，它很快，并且对着色器及纹理支持良好。但是，Phoria 基于软件渲染而设计，因此它的 3D 能力受限。某些 3D 功能光靠软件是几乎不可能实现（或者实现好）的。

A.4.2　3D游戏引擎

现在在市场上出现了许多 WebGL 游戏引擎。它们是构建游戏和复杂 3D 应用的好选择，但对于简单 3D 开发项目来说就有点杀鸡用牛刀了（关于这一点，请看 A.4.3 节）。除非另有说明，否则这里列出的游戏引擎都是开源的。

- playcanvas（http://www.playcanvas.com/）

  位于伦敦的 playcanvas 开发了一个强大的引擎及基于云的编辑工具。编辑工具的功能有支持团队开发的实时协同场景编辑，GitHub 和 Bitbucket 集成，一键发布到社交媒体网络上。在我编写本书的时候，playcanvas 发布了客户端引擎的源码，但它没有发布许可条款。

- Turbulenz（http://biz.turbulenz.com/）

  Turbulenz 是一个非常强大的、开源、无授权费的游戏引擎，并打包成了一个可下载的 SDK。该公司对想要在它的网络（https://ga.me/）上发布作品的开发者收取费用。Turbulenz 拥有最复杂的 API，有大量类，学习曲线陡峭。显然它是提供给有经验的游戏开发者的。Turbulenz 提供了开源的客户端库，保留了系统的其他部分（服务器、虚拟货币等）来创造收入。

- Goo Engine（http://www.gootechnologies.com/）

  Goo 最近发布了需要邀请的 beta 版本引擎和内容创建工具。作为 Goo Engine 的配套，这家公司提供了容易使用的内容创建前端，目标用户是主流 Web 开发者。在本书编写期间，Goo 还没开源。

- Verold（http://www.verold.com/）

  一个轻量级的 3D 交互内容发布平台，由位于多伦多的 Verold 公司开发。公司将其描述为“一个没有插件的、可扩展的系统，使用简单的 JavaScript 来让业余爱好者、学生、教师、可视化通信专家以及 Web 市场人员来简单集成 3D 动画内容到他们的 Web 页面中”。和 Goo 一样，Verold 面向一般 Web 图形开发，引擎复杂但前端简单。本书编写期间，Verold 还没有开源。

- Babylon.js（http://www.babylonjs.com/）

  Babylon.js 是微软员工 David Catuhe 开发的一个个人项目。它是一个易于使用的引擎，在功能及易用性上介于 Three.js 和硬核游戏引擎之间。它的演示网站上展示的应用跨度范围大，从太空射击到建筑浏览。

- KickJS（http://www.kickjs.org/）

  Morten Nobel-Jørgensen 创建的一个开源游戏引擎和渲染库，这个项目是从他的学术工作中发展而来的。在开发和支持上，KickJS 似乎比不上这里列出的其他游戏引擎。但将它包括在学习列表中主要是因为，在介绍的所有游戏引擎中，KickJS 最严格地遵循现代游戏引擎设计的最佳实践。

A.4.3 3D展示框架

加快开发 3D 的需求导致了几个实验展示框架的出现。和游戏引擎不同，这些框架的重点是快速简单地在页面中嵌入图形，用于数据可视化、产品展现、简单的动画等。

- Voodoo.js（http://www.voodoojs.com/）

  Voodoo.js 的目标是让创建 3D 内容变得简单，并容易嵌入到页面中。Voodoo.js 有一个极其简单的 API，用来添加 3D 模型到页面中：只需提供模型的 URL、一个 DIV 元素的 id 和一些配置参数，你就能在页面中拥有 3D 了。Voodoo.js 的功能主要是页面中的简单模型查看，但仅就这一个用途来说，它就相当不错了。

- tQuery（http://jeromeetienne.github.io/tquery/）

  tQuery 由 Jerome Etienne 创建，他同时运营着流行的的博客 Learning Three.js（http://learningthreejs.com/）。tQuery 模仿 jQuery 库，目的是提供“Three.js 的功能 + jQuery API

的可用性”，也就是说，用一个非常简单的 API 来操作 Three.js 场景图。它使用了一个链式函数编程模式，支持通过回调进行高级别的交互。使用 tQuery 可以节省很多行 Three.js 代码的编写。或许把 tQuery 称为框架并不正确，因为它更像是一个本着 jQuery 精神的无侵入库。tQuery 可以为想节省键盘操作的 Three.js 开发者节省大量时间。

- PhiloGL（http://www.senchalabs.org/philogl/）

  PhiloGL 是一个实验性的框架，由数据可视化专家 Nicolas Garcia Belmonte 在 Sencha 公司的实验室工作时创建。PhiloGL 的目标是“让 WebGL 编程像使用任何主流框架开发那样有趣和简单”。Garcia 在他的博客文章（http://www.sencha.com/blog/introducing-philogl-a-webgl-javascript-library-from-sencha-labs/）中描述了他的设计理念。尽管这个框架是实验性的，但它值得一看。Sencha 公司开发了世界一流的用户界面框架，知道如何用 HTML5 创建有效的用户界面。PhiloGL 网站包含了一些可工作的例子，移植了 Learning WebGL（http://www.learningwebgl.com/）的全部教程。

- Vizi（https://github.com/tparisi/Vizi）

  我自己设计的一个展示框架，它凝结了我开发早期 3D 框架和引擎（比如 VRML 和 X3D）的多年经验。Vizi 采用了当前游戏引擎的最佳实践，最值得一提的是它使用组件化和聚合来构建高层次的功能，而不是基于类的继承。Vizi 的目标是使开发者易于快速构建有趣的 3D 应用。和 Voodoo.js 一样，Vizi 允许开发者用几行代码将一个模型放到页面中，但是它还提供了完整的高层次 API 来添加交互、动画、行为到场景中的任意元素上。

A.4.4 3D编辑工具

1. 传统建模和动画软件

Autodesk（http://www.autodesk.com/）提供了一系列 3D 建模和动画软件，它们的价格都比较高。目前 Autodesk 开始提供软件的学习和试用版，它们值得一试。

除了 Autodesk 的专业套件，还有几个免费或不太贵的软件可用来创建 3D 内容，介绍如下。

- Blender（http://www.blender.org/）

  一个免费、开源、跨平台的 3D 创建工具套件，Blender 可以运行在所有主流操作系统上，以 GNU GPL 的方式授权。Blender 由荷兰软件开发者 Ton Roosendaal 创建，由 Blender 基金会（荷兰的一个非营利性组织）负责维护。Blender 非常流行，估计有两百万用户，从业余 / 学生级别用户到专业设计师和工程师。

- SketchUp（http://www.sketchup.com/）

  SketchUp 是一个易于使用的 3D 建模程序，用在建筑、工程和一些游戏开发中。你可以在它的网站找到免费和不算贵的专业版本。

- Poser（http://poser.smithmicro.com/）

  一个用于角色动画的中级 3D 工具。Poser 和 SketchUp 一样，价格吸引人，并针对日常的内容创建用户。它有一个直观的用户界面来摆放造型和动画角色。Poser 还有一个庞

大的库，包含已经建好模型、骨架且带完整贴图的人物和动物角色，以及一系列背景场景、道具、载体、相机和光照设置。Poser 可以用来创建照片级的静态渲染和实时动画。

2. 基于浏览器的集成环境

有了云计算和在 WebGL 渲染的能力，我们看到了一种新型创建工具：在浏览器中的 3D 集成环境。以下工具还在早期开发中，但很有前途。

- Goo Create（http://www.gootechnologies.com/）

  前面讨论的 Goo Engine 搭配了一个易于使用的内容创建前端——Goo Create。Goo Create 的目标用户是主流 Web 开发者。它还有几个预先创建好的模型和动画，可以依此进行开发学习。

- Verold Studio（http://www.verold.com/）

  Verold Studio 是一个在浏览器中运行的 3D 内容创建工具和开发环境，它包含了之前介绍过的 Verold 游戏引擎。

- Sketchfab（http://sketchfab.com/）

  Sketchfab 是一个实时在线发布和分享交互式 3D 内容的 Web 服务，它不需要插件。只需要点击几下，艺术家就能上传一个 3D 模型（可以是几种格式）到 Sketchfab 网站上，来获得 HTML 代码以分享一个部署在 Sketchfab 网站上的内嵌模型视图。

- SculptGL（https://github.com/stephomi/sculptgl）

  一个免费和开源的 Web 实体建模工具，有非常易用的界面来创建简单的雕塑模型。SculptGL 可以导出到多种格式，并直接发布到 Verold 及 Sketchfab 上。

A.4.5 动画框架

现今的应用应该使用 `requestAnimationFrame()` 来让内容动起来。为了保证这个功能得到跨浏览器支持，可以使用 Paul Irish 的强大 polyfill（http://paulirish.com/2011/requestanimationframe-for-smart-animating/）。

对于简单的补间动画，Tween.js（https://github.com/tweenjs/tween.js）是 Soledad Penadés 创建的流行开源补间工具。

对于关键帧动画，在 Three.js 中有些内建的类，而在随项目发布的例子中还有更多。随着更多在线工具的出现，以及类似 glTF 那样对 Web 友好格式的成熟，这个领域还在不断进化。

A.4.6 调试和分析WebGL应用

新版的浏览器自带了各种 WebGL 调试和分析工具。AGI（Cesium 的开发商，Cesium 是一个基于 WebGL 的虚拟地球和地图引擎）的图形架构师 Patrick Cozzi，写了一篇浏览器内置 WebGL 工具的优秀综述（http://www.realtimerendering.com/blog/webgl-debugging-and-profiling-tools/）。

A.4.7　移动3D开发资源

添加触摸支持是创建引人入胜的移动 3D 应用的关键。浏览器触摸事件规范可以在 W3C 推荐页面找到（http://www.w3.org/TR/2013/REC-touch-events-20131010/）。

Android 的开发者页面（http://developer.android.com/guide/webapps/index.html）包含了关于开发基于 HTML5 的 Web 应用的完整信息。

Amazon 有一个用来发布 Web 应用的广泛的系统，包括一个 Web App Tester 应用，它用于基于 Android 的 Kindle Fire OS（https://developer.amazon.com/sdk/webapps/tester.html），还有一个为了打包和发布最终应用的应用分发入口（https://developer.amazon.com/sdk/webapps/manifest.html）。

对于不原生支持 WebGL 的环境，比如 iOS，还有混合技术可以结合 HTML5/JavaScript 和原生代码来构建应用。尽管 Adobe 的 PhoneGap（http://phonegap.com/）是主流的移动混合库，但它现在还不支持 WebGL。为了让 iOS 支持 WebGL，可以使用下面混合框架中的一个。

- CocoonJS（http://www.ludei.com/tech/cocoonjs）

  CocoonJS 可以运行在 Android 和 iOS 上，它提供了一个简单易用的应用容器来运行 HTML5 和 JavaScript 代码，隐藏了底层系统的细节。CocoonJS 提供了对 Canvas、WebGL、Web Audio、Web Sockets 等的实现。它还有一个云端的项目编译系统，所以你要做的仅仅是登录和编译你的项目。开发者不需要了解如何使用复杂的原生平台工具，比如 iOS 的 Xcode，来创建原生应用。CocoonJS 是闭源项目，由它位于旧金山的开发商 Ludei 严格控制。

- Ejecta（http://impactjs.com/ejecta）

  Ejecta 是一个开源库，提供了许多和 CocoonJS 一样的功能，但只支持 iOS。Ejecta 诞生自 ImpactJS，一个 HTML5 游戏引擎项目。Ejecta 更加 DIY，需要开发者对 Xcode 和原生平台 API 有相当的了解。Ejecta 是开源的。

A.5　3D文件格式规范

3D 文件格式分为三种类型：模型格式，用于表示单个对象；动画格式，用于动画关键帧和角色；全功能格式，支持整个场景，包括多个模型、变换层级结构、相机、光源、动画。3D 文件格式有很多种，这里不能一一列举。

以下 3D 格式最适合 Web 应用开发。

A.5.1　模型格式

- Wavefront OBJ（http://en.wikipedia.org/wiki/Wavefront_.obj_file）。
- STL——基于文本的 3D 打印文件格式（http://en.wikipedia.org/wiki/STL_%28file_format%29）。

A.5.2　动画格式

- id Software 的 MD2（http://tfc.duke.free.fr/coding/md2-specs-en.html）和 MD5（http://tfc.duke.free.fr/coding/md5-specs-en.html）都是角色动画格式。
- BioVision 公司的 BVH 动画格式，用于动作捕捉（http://research.cs.wisc.edu/graphics/Courses/cs-838-1999/Jeff/BVH.html）。

A.5.3　完整场景格式

- VRML（http://www.web3d.org/standardsvrml/）和 X3D（http://www.web3d.org/standards-x3d/）——Web 最初的 3D 格式。
- COLLADA（http://www.khronos.org/files/collada_spec_1_4.pdf）——数字资源交换格式。
- glTF（http://gltf.gl/）——图形库传输格式。

A.6　相关技术

3D 开发并不脱离现实。你也许想要把其他有趣的 Web 技术整合到你的 3D 项目中。这里列出了其中一些。

A.6.1　指针锁定API

对于全屏的 3D 应用，比如游戏，你也许希望对鼠标输入提供比传统 DOM 窗体事件提供的更精细的控制。为此，浏览器最近引入了指针锁定 API，它让开发者可以隐藏鼠标光标并获取游戏开发所需类型的底层鼠标活动事件。

Google 的 John McCutchan 为指针锁定 API 编写了一套很好的说明（http://www.html5rocks.com/en/tutorials/pointerlock/intro/）。

你可以在 W3C 的官网上找到指针锁定 API 的最新 W3C 规范（http://www.w3.org/TR/pointerlock/）。

A.6.2　页面可见性API

每秒 60 帧的 3D 应用会耗费机器性能。如果一个应用的标签页或窗体当前不可见，那么其场景就无需被渲染。但应用可能仍需要在后台计算结果，尽管不需要那么频繁。最近的浏览器支持一个新的特性，页面可见性 API（Page Visibility API），它让开发者知道页面或标签页何时是不可见的，以便调整执行来节省机器资源。

在谷歌开发者社区上有一篇关于页面可见性 API 的优秀概述（https://developers.google.com/chrome/whitepapers/pagevisibility）。

你可以在 W3C 的官网上找到页面可见性 API 的最新 W3C 规范（http://www.w3.org/TR/pointerlock/）。

A.6.3 WebSockets和WebRTC

如果你在开发一个多人互动 3D 游戏、虚拟世界或者是实时协同的应用，你可能需要实现客户端和服务端之间的通信。WebSockets 和 WebRTC 是用于实现这类功能的两项主要技术。

WebSockets（更正式地说，是 WebSocket 规范）是浏览器对 TCP/IP 协议的标准实现。它可以用于客户端和服务端之间的双工通信。TCP/IP 本身不是为实时通信设计的，因此 WebRTC（稍后介绍）可能会更合适，这取决于应用的需求。这里有一篇关于 WebSockets 的教程（http://net.tutsplus.com/tutorials/javascript-ajax/start-using-html5-websockets-today/），你还可以访问 WebSockets 项目的主页（http://www.websocket.org/）。

WebRTC 是在 Web 客户端和服务器间发送实时消息的标准。它比 WebSocket 协议更适合多个用户间发送消息，因为它就是设计用来进行实时通信的。教程可以参考 http://www.html5rocks.com/en/tutorials/webrtc/basics/。项目主页由 Google 维护，地址是 http://www.webrtc.org/；当前 W3C 推荐在 http://www.w3.org/TR/webrtc/。

A.6.4 Web Workers

Web Workers（http://www.w3.org/TR/webrtc/）支持在 JavaScript 中多线程开发。3D 应用可以通过在背景线程做某些操作而受益，比如加载模型或运行物理模拟。将这些操作放到后台，应用可以保证用户界面始终可响应，即便应用正在处理计算密集型操作。

使用 Web Workders 有些需要注意的事情，比如在线程间传递内存对象。在 HTML5 Rocks 上有一篇关于这个细节的优秀文章（http://updates.html5rocks.com/2011/12/Transferable-Objects-Lightning-Fast）。

A.6.5 IndexedDB和FileSystem API

3D 文件可以变得很大。对于你的项目而言，你可能要考虑使用新的 HTML5 技术存储数据到用户硬盘，来帮助节省下载开销。浏览器缓存不可靠，因为它们没那么大，而且不受应用控制——用户可以随时清掉缓存，或其他 Web 数据会将你应用的内容清出缓存。

Adobe 的开发者布道师及本书的技术审阅人 Ray Camden，提到了可以使用浏览器数据库 API IndexedDB 来存储本地数据的想法。他写了一篇关于这个话题的开发富 SVG 应用的文章（http://www.raymondcamden.com/index.cfm/2013/2/5/Playing-with-SVG-and-JavaScript）。你可以在 http://www.w3.org/TR/IndexedDB/ 找到 IndexedDB 规范。

然而 IndexedDB 并不是一个文件系统，它是一个数据库 API。如果你想使用文件系统类的 API 来存储和获取用户计算机上的内容，你很幸运，有一个实验性的 API 叫 FileSystem API（http://www.w3.org/TR/file-system-api/）。有了这个 API，Web 应用就能在用户硬盘上读取和写入文件和目录层次结构。HTML5 Rocks 上有一篇优秀的教程（http://www.html5rocks.com/en/tutorials/file/filesystem/）。注意 FileSystem API 目前只在 Opera 和桌面 Chrome 中得到支持。另外注意不要和 File API（http://www.w3.org/TR/FileAPI/）混淆，它只允许读取本地文件系统。

作者介绍

Tony Parisi 是一位企业家和职业 CTO/ 架构师。他开发了国际标准和协议，创造了卓越的软件产品，并创立和出售过技术公司。Tony 对创新充满热情，但他渴望为广大潜在读者带来最酷和最有趣的东西。

Tony 最为知名的应该是他是 Web 3D 标准的先驱。他是网络 3D 图形 IOS 标准 VRML 和 X3D 的联合创造者。他还联合开发了 SWMP，一个用于多用户虚拟世界的实时消息协议。Tony 是 WebGL Meetup 的联合主席及 Rest3D 工作组的创始人，他持续构建围绕 3D 创新的社区。

Tony 现在是一个在线游戏创业公司的合伙人，同时还为旧金山湾区的客户提供开发社交游戏、虚拟世界和基于位置服务的技术支持。

封面介绍

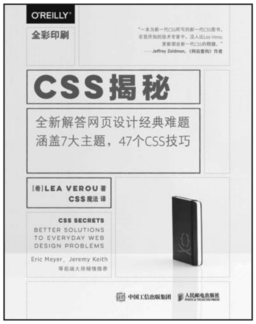
本书封面上的动物是麦奎因大鸨（波斑鸨），一种生活在中东及西南亚地区的大鸟。它以将军托马斯•麦奎因的名字命名。麦奎因是 19 世纪的一名英国士兵，驻扎在印度。他还是自然历史标本收藏家，并向大英博物馆捐赠了他射下的一只大鸨，这种鸟于 1832 年以他的名字命名。

麦奎因大鸨在干旱沙地地区生存和繁殖，以种子、植物芽和昆虫为食。一般长 2 英尺，翼展 55 英寸，雌性的体型稍小。它们的羽毛是浅棕色的，脖子上有黑色条纹，下腹为白色，头上和脖子上的蓬松羽毛会在交配的时候呈扇形散开。它们不常发出叫声，在地面打洞筑巢，每次生 2~4 个蛋。

这个物种（及它的近亲翎领鸨）变得越来越稀有，它们因受到驯鹰者的欢迎而被过度捕猎。虽然某些中东地区的领导，包括沙特阿拉伯及阿联酋王室，都在近些年对这种鸟类采取了保护措施，但它依然濒临灭绝。

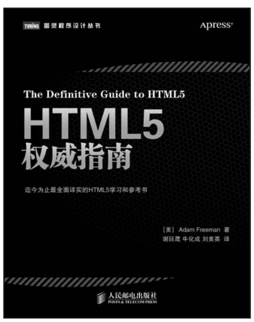
封面图片来自约翰逊的 *Natural History*。

延 展 阅 读



全新解答网页设计经典难题，涵盖 7 大主题、47 个 CSS 技巧
Eric Meyer、Jeremy Keith 等前端大师倾情推荐

书号：978-7-115-41694-0
定价：99.00 元



迄今为止最全面详实的网页设计参考书
精彩呈现 500 多个实战代码示例及主流浏览器实现效果图
贴心汇聚 HTML5 和 CSS3 中所有属性、元素和函数的简明参考表

书号：978-7-115-33836-5
定价：129.00 元



在本书中，我们要直面当前JavaScript开发者不求甚解的大趋势，深入理解语言内部的机制。本书既适合JavaScript语言初学者阅读，又适合经验丰富的JavaScript开发人员深入学习。

书号：978-7-115-38573-4
定价：49.00 元



网络时代，用户体验的重要性毋庸置疑，动画在这一过程中的重要性也明显提升。如何在不分散用户注意的情况下达到动画设计加强页面目的的效果，已经成为优秀的用户界面设计师和Web开发人员孜孜以求的目标。本书将为此提供必备的知识。

书号：978-7-115-41012-2
定价：39.00 元

O'REILLY®

HTML5与WebGL编程

本书介绍如何使用HTML5相关技术，如CSS3和新兴的Web图形标准WebGL，来创建具有高性能、震撼视觉效果的3D Web应用。书中内容分为两部分——基础知识和应用开发技术，不但提供了全面的理论介绍，还包括从简单3D产品可视化到沉浸式游戏及交互训练系统的实践，适合转向3D开发的Web开发人员阅读。

- **探索HML5 API及创建3D Web图形的相关技术，包括WebGL、Canvas和CSS**
- **使用流行的JavaScript 3D渲染和动画库Three.js及Tween.js**
- **研究3D内容创作流程，创建杀手级3D内容的建模和动画工具**
- **介绍构建3D应用的游戏引擎和框架，包括作者的Vizi框架**
- **使用示例及支持代码，创建有多个物体和复杂交互的3D场景**
- **分析移动浏览器中的WebGL 3D应用会遇到的问题**

"Tony Parisi在早期就引领了将3D带入Web的革命。这本新书深入研究了底层技术，并提供了在现代浏览器中使用前沿3D技术的经验。"

——Neil Trevett
NVIDIA移动内容副总裁，
Khronos小组主席

Tony Parisi，Web 3D标准的先驱、企业家、CTO、架构师。VRML和X3D语言的联合作者，这两者已经成为Web 3D图形的ISO标准。另著有《WebGL入门指南》。

WEB DEVELOPMENT

封面设计：Karen Montgomery　**张健**

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机 / Web开发

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

ISBN 978-7-115-42133-3

9 787115 421333 >

ISBN 978-7-115-42133-3
定价：79.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks